The Descartes Meta-Model

Samuel Kounov, Fabian Brosig, Nikolaus Huber

Descartes Research Group
Institute for Program Structures and Data Organization
Karlsruhe Institute of Technology (KIT), Germany

kounev@kit.edu brosig@kit.edu huber@kit.edu

February 6, 2013

v0.81
## Contents

1 **Introduction** .......................................................... 6  
   1.1 Motivation ............................................................ 6  
   1.2 Design-time vs. Run-Time Models .............................. 8  
   1.3 Descartes Meta-Model (DMM) ..................................... 11  
   
   1.3.1 Meta-Model Overview .......................................... 11  
   1.3.2 Summary of Supported Features and Novel Aspects ....... 13  
   1.4 Self-Aware System Architectures ............................... 14  
   1.5 Outline ............................................................. 15  

2 **Background** ............................................................. 16  
   2.1 Performance Modeling Approaches .............................. 16  
   
   2.1.1 Architecture-level Performance Models .................... 16  
   2.1.2 Palladio Component Model (PCM) ............................ 17  
   2.2 Modeling Run-time System Adaptation ......................... 20  
   
   2.2.1 Abstraction Levels ............................................ 20  
   2.2.2 Languages for Adaptation Control Flow ................... 20  
   2.2.3 Configuration Space .......................................... 21  

3 **Online Performance Prediction Scenario** ......................... 22  
   3.1 Setting ............................................................. 22  
   3.2 SPECjEnterprise2010 .............................................. 23  
   3.3 Experimental Environment ....................................... 24  

4 **Application Architecture and Resource Landscape** ............. 26  
   4.1 Motivation and Background ....................................... 26  
   
   4.1.1 Component Interfaces and Component Composition .......... 26  
   4.1.2 Component Context ............................................ 28  
   4.1.3 Service Abstraction Levels ................................... 31  
   4.1.4 Parameter Dependencies ...................................... 33  
   4.1.5 Virtualized Resource Landscape .............................. 36  
   4.2 Concepts ........................................................... 38  
   
   4.2.1 Service Behavior Abstractions for Different Levels of Granularity 39  
   4.2.2 Parameter and Context Dependencies .......................... 41  
   4.2.3 Dynamic Virtualized Resource Landscapes .................. 47  

5 **Model-based System Adaptation** ................................ 54  
   5.1 Motivation and Background ....................................... 54
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2 Concepts</td>
<td>56</td>
</tr>
<tr>
<td>5.2.1 Adaptation Points Model</td>
<td>56</td>
</tr>
<tr>
<td>5.2.2 Adaptation Language</td>
<td>58</td>
</tr>
<tr>
<td>Index</td>
<td>68</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 Degrees-of-Freedom and performance-influencing factors in a modern IT system. . . . . 7  
1.2 Structure of the Descartes Meta-Model (DMM). . . . . . . . . . . . . . . . . . . . . . . 11  

2.1 Components providing and requiring interfaces. . . . . . . . . . . . . . . . . . . . . . . 18  
2.2 Assembly of a composite component. . . . . . . . . . . . . . . . . . . . . . . . . . . . 18  
2.3 RD-SEFF of service with signature `execute(int number, List array)` (cf. [1]). . . . . 19  

3.1 Online performance prediction scenario. . . . . . . . . . . . . . . . . . . . . . . . . . . 22  
3.2 SPECjEnterprise2010 architecture [2]. . . . . . . . . . . . . . . . . . . . . . . . . . . . 23  
3.3 Experimental environment. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24  

4.1 `InterfaceProvidingRequiringEntity` in the PCM Meta-Model (cf. [3]). . . . . . . . . 27  
4.2 `ComposedStructure` in the PCM Meta-Model (cf. [3]). . . . . . . . . . . . . . . . . . . 27  
4.3 Example of a nested composed structure showing the component types. . . . . . . . . 28  
4.4 Example of a nested composed structure showing the composition of the components. . 28  
4.5 Performance-influencing factors of a component (cf. [3]). . . . . . . . . . . . . . . . . . 29  
4.6 Adding a node to a customer’s cluster. . . . . . . . . . . . . . . . . . . . . . . . . . . . 30  
4.7 Two clusters for two different customers. . . . . . . . . . . . . . . . . . . . . . . . . . . 30  
4.8 Example: Deployment-specific resource demands. . . . . . . . . . . . . . . . . . . . . . 31  
4.9 Supply Chain Management component. . . . . . . . . . . . . . . . . . . . . . . . . . . . 32  
4.10 SCM component internals. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 33  
4.11 Response time statistics of `sendPO`. . . . . . . . . . . . . . . . . . . . . . . . . . . . 33  
4.12 Manufacturing component. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34  
4.13 `Dealer` and `Manufacturing` components. . . . . . . . . . . . . . . . . . . . . . . . . 35  
4.14 WorkOrder data structure. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35  
4.15 `scheduleManufacturing` statistics. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36  
4.16 Main types of data center resources. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 37  
4.17 Different resource layers and their influence on the performance. . . . . . . . . . . . . 38  
4.18 Service behavior abstraction. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40  
4.19 Black-box behavior (a) and coarse-grained abstractions (b). . . . . . . . . . . . . . . 40  
4.20 Fine-grained behavior abstraction. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40  
4.21 Modeling parameter dependencies. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 42  
4.22 Model variables. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 43  
4.23 Call parameter hierarchy. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44  
4.24 Call parameters. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 44  
4.25 Relationships between `InfluencedVariables` and `InfluencingParameters`. . . . . . 45  
4.26 `ExplicitCharacterization` and `EmpiricalCharacterization`. . . . . . . . . . . . . . . 47  
4.27 Resource landscape meta-model. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
4.28 Different runtime environment classes. ................................. 49
4.29 Example resource landscape model instance with different container types. ................................. 50
4.30 Types of resource configurations (a) and the container repository (b). ................................. 50
4.31 An example container repository with three container templates. ................................. 52
4.32 The deployment model. ................................................... 53
5.1 Interaction of the system, the system models and the S/T/A adaptation language. ................................. 55
5.2 Adaptation points model. ................................................... 56
5.3 Example of an adaptation points model instance with three different adaptation points. ................................. 57
5.4 Hierarchy of adaptation steps. ............................................ 59
5.5 Adaptation language meta-model. ........................................ 60
5.6 Actions for the running example. ........................................ 61
5.7 Tactics for the running example. ........................................ 63
5.8 Strategies using tactics and assigning weights. ........................................ 64
Chapter 1

Introduction

This technical report introduces the Descartes Meta-Model (DMM), 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. DMM is designed to serve as a basis for self-aware resource management\(^1\) \cite{4, 5} 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 DMM 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 DMM 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 Meta-Model (DMM) 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 \cite{6, 7, 8}. 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 \cite{9}. 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 DMM is focused on performance, hereafter we will speak of online performance prediction [4, 5].

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., [10, 11, 12, 13]) 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., [1, 14, 15]). 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 [16] 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 unavailability of a realistic production-like testing environment and the typical lack of complete implementations of all system components.

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

The Descartes Meta-Model (DMM) consists of different sub-meta-models depicted in Figure 1.2. Each sub-meta-model has its own specific requirements and features. Section 1.3.1 gives a high-level overview of DMM covering its organization and structure, and gives examples for possible application scenarios. Section 1.3.2 summarizes and explains the supported features and novel aspects of DMM. In the remainder, we use the terms meta-model and sub-meta-model synonymously for convenience reasons.

1.3.1 Meta-Model Overview

The Descartes Meta-Model (DMM) is a new architecture-level modeling language. It is developed to model modern dynamic IT systems, infrastructures and services with the target to use these model instances for self-aware system adaptation at run-time.

Figure 1.2 shows a high-level overview of DMM and its four different sub-meta-models, each having its specific purpose and features. The Resource Landscape meta-model describes the physical and logical resources (e.g., virtualization and middleware layers) provided by modern dynamic data centers. The Application Architecture meta-model describes the performance-relevant service behavior of the applications executed on the resources. Together, these two meta-models form the System Architecture QoS Model which can already be applied in scenarios like capacity planning or trade-off analysis at design-time. Additionally, the Adaptation Points meta-model defines so-called adaptation points to describe the parts of the System Architecture QoS model that are adaptable at run-time, i.e., while the system is up and running. The adaptation points meta-model spans the configuration space of the modeled system, i.e., it describes the valid states a system can have at run-time. Thereby, the adaptation points model can
also be used in scenarios focussing on describing and evaluating system scalability and elasticity. Finally, DMM provides an Adaptation Process meta-model, a modeling language to describe the processes implemented in systems to adapt to changes in the environment. This model is intended to describe scenarios like self-aware QoS management and resource efficiency at run-time.

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

**Resource Landscape** Modern dynamic data centers consist of distributed physical resources as well as multiple logical resource layers that can be reallocated at run-time. The resource landscape meta-model describes the resources and their structural order, i.e., it reflects the static view of distributed dynamic data centers. The purpose of these resource landscape models is to provide more detailed structural information than existing modeling approaches. For example, current approaches do not provide means to model the layers of the component execution environment (e.g., the virtualization layer) explicitly [17]. The main benefit of our resource landscape meta-model is the increased model expressiveness to exploit this information in the adaptation processes to improve decision making and find better solutions.

The main features of the resource landscape meta-model are the means to 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. Furthermore, the meta-model provides means to describe the layered execution environment, increasing the flexibility and reuse of reoccurring container types. Our modeling approach is generic, covering all types of resources including active resources like CPU and HDD, as well as passive resources like thread pools or database connections. Additionally, the influences of the individual layers on the system’s performance, the dependencies among these influences and the resource allocations at each layer can be captured as part of the models. This is necessary in order to be able to predict at run-time how a change in the execution environment (e.g., modifying resource allocations at the VM level) would affect the system performance.

**Application Architecture** The application architecture is modeled as 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, four factors can be taken into account. Obviously, the component’s implementation affects its performance. Additionally, the component may depend on external services whose performance has to be considered as well. Furthermore, both ways the component is used, i.e., its usage profile, and the execution environment in which the component is running are taken into consideration.

For the description of the performance behavior of a component our meta-model supports having multiple (possibly co-existing) behavior abstractions at different levels of granularity. The behavior descriptions range from a “black-box” representation capturing the view from the perspective of a service consumer without any additional information about the service behavior, to a “fine-grained” representation allowing to describe component-internal control flow including information about component-internal resource consuming actions.

The behavior of software components is often dependent on parameters that are not available as input parameters passed upon service invocation. 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. We provide novel modeling abstractions and concepts for expressing and resolving these parameter and context dependencies. For instance, we support probabilistic characteri-
zations of parameter dependencies that are based on monitoring data. Furthermore, given that the same software component might be used multiple times in different contexts, we introduce a scope of a parameter indicating where the parameter is unique. Thus, we can distinguish situations where measurements of a parameter can be used interchangeably among component instances or are not transferable across certain component boundaries.

**Adaptation Points** Today’s distributed data centers are increasingly dynamic and offer high flexibility for adapting systems at run-time. This has to be reflected in the models of such systems to analyze the impact of system adaptation and find good adaptation actions at run-time. The adaptation points meta-model is an addition to the system architecture QoS model of DMM. The purpose of the adaptation points meta-model is to describe the degrees-of-freedom of the dynamic system, i.e., which parts of the resource landscape and application architecture meta-models are variable and can be adapted during run-time. In other words, this model reflects the boundaries of the system’s configuration space, i.e., it defines the possible valid states of the system architecture. This avoids modeling each of the various configurations the system might have at run-time. However, it is not intended to specify how to change the model instance or even the system, i.e., the actual change itself is implemented in the adaptation process using the adaptation points model.

The meta-model provides several ways to describe adaptation possibilities, e.g., simple model parameter ranges by specifying its minimum and maximum values as well as complex constraints using OCL expressions.

**Adaptation Process** In general, complex adaptations to changes in the system environment are still largely performed manually by humans. Furthermore, automated or autonomous adaptation processes are usually highly system specific and it is a challenge to abstract from system details to enable the reuse of adaptation strategies. Our adaptation process meta-model provides means to describe system adaptation processes at the system architecture level in a generic, human-understandable and reusable way. The meta-model explicitly defines a set of modeling abstractions to describe strategies, tactics and actions with the flexibility to model the full spectrum of self-adaptive mechanisms, abstracting from system specific details.

Our approach has several important features. It distinguishes high-level adaptation objectives (strategies) from low-level implementation details (adaptation tactics and actions). Thereby, it explicitly separates platform adaptation operations from system adaptation plans. We argue that separating these concerns has the benefit that system designers can describe their knowledge about system adaptation operations, independently of how these operations are used in specific adaptation scenarios. Additionally, given the fact that the knowledge about system adaptations is described using a meta-model with explicitly defined semantics, this knowledge is machine-processable and can thus be easily maintained and reused in common adaptation processes in dynamic systems like cloud environments.

### 1.3.2 Summary of Supported Features and Novel Aspects

The Descartes Meta-Model (DMM) 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. DMM 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 DMM 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, DMM 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 DMM 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.

1.4 Self-Aware System Architectures

As mentioned above, a major application of the Descartes Meta-Model (DMM) is to serve as a basis for self-aware resource management during operation. In this section, we explain in more detail what exactly is meant by self-awareness in this context.

DMM is a major part of our broader long-term research effort aimed at developing novel methods, techniques and tools for the engineering of self-aware system architectures [4, 5]. 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 systems should possess:

1. **Self-reflective**: aware of their software architecture, execution environment and hardware infrastructure on which they are running, as well as of their 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 service deployment and/or resource allocations),
3. **Self-adaptive**: proactively adapting as the environment evolves in order to ensure that their operational goals are continuously met.

The Descartes Meta-Model (DMM) 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

---

\[\text{http://www.descartes-research.net}\]
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 workloads). 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 [20]. Each time an online QoS query is executed, it is processed based on the online system architecture models (DMM 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., [21]) 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 4.2. The report concludes with a discussion of the differences between DMM and PCM, open issues as well as limitations and provides an outlook on future work in Chapter ??.
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) [1], a meta-model for design-time performance analysis of component-based software architectures, which has inspired some of the core elements of DMM.

A survey of model-based performance prediction techniques was published in [22]. A number of techniques utilizing a range of different performance models have been proposed including product-form queueing networks (e.g., [10]), layered queueing networks (e.g., [12]), queueing Petri nets (e.g., [23]), stochastic process algebras [24], statistical regression models (e.g., [25]) and learning-based approaches (e.g., [26]). 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 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 models.

\(^1\)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 [27] and its successor the UML MARTE profile [15], both of which are extensions of UML as the de facto standard modeling language for software architectures. Other proposed meta-models include CSM [28], PCM [1], SPE-MM [14], and KLAPER [29]. A recent survey of model-based performance modeling techniques for component-based systems was published in [16].

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 [16] 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 [30]. 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 [31].

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) [1]. 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 DMM.

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
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.
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)` [1] 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 DMM’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 [32, 33], 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 [34] to specify policy types for autonomic computing or especially in [35], 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 [36], 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) [37] or the Business Process Execution Language (BPEL) [38]) 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 [32].

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 [39] 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. [40] 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 of 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 [41] 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 DMM, 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 Meta-Model (DMM). 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.
3.2 SPECjEnterprise2010

SPECjEnterprise2010 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

\[\text{http://www.spec.org/jEnterprise2010/}\]
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 Experimental 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

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

Application Architecture and Resource Landscape

In this chapter, we present the application architecture and resource landscape sub-meta-models of DMM, i.e., collectively used to define a system architecture QoS model. The current version of DMM 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 DMM is focussed on performance, in the rest of this section, we mostly speak of performance instead of QoS in general.

4.1 Motivation and Background

We first provide an overview on the motivating background behind the application architecture and resource landscape sub-meta-models of DMM. Starting with the component model of PCM in Section 4.1.1, highlighting the interface concept and the concept of component composition, Section 4.1.2 discusses the performance-relevant aspects of a component with regard to its deployment context from several different perspectives. Section 4.1.3 motivates the need for modeling service behavior at different abstraction levels and Section 4.1.4 motivates the relevance of parameter dependencies. Finally, in Section 4.1.5, we describe the aspects of the resource landscape that need to be modeled explicitly to enable self-aware run-time performance management.

4.1.1 Component Interfaces and Component Composition

We start by introducing the meta-model underlying the component model of PCM which was also used as a basis for DMM’s component model. We describe how interfaces are mapped to their providing/requiring components and how composite components are assembled.

4.1.1.1 Interfaces as First-Class Entities

In PCM, interfaces are modeled explicitly, i.e., they are first-class entities that can exist on their own. Consequently, a component does not have an interface, but it may provide and/or require some interfaces. The connection between components and interfaces is specified using so-called roles. A component can take two roles relative to an interface. It can either implement the functionality specified in the interface (provided role) or it can require that functionality (required role).

Figure 4.1 shows the corresponding meta-model. An InterfaceProvidingEntity may have ProvidedRoles that refer to an Interface. An InterfaceRequiringEntity is modeled accordingly
with RequiredRoles. BasicComponents (i.e., atomic components) and CompositeComponents both can require and provide interfaces. A CompositeComponent also inherits from type ComposedStructure. The latter type is described in the next section.

### 4.1.1.2 Composed Structure

Figure 4.2 shows how a ComposedStructure is assembled. A ComposedStructure may contain several AssemblyContexts which themselves each refer to a BasicComponent. Each AssemblyContext thus represents a child component instance in the composite. An AssemblyConnector connects two child component instances with a required and a provided role, representing a connection between a provided role of the first component and a required role of the second component. Connectors from the child component instances to the composite component boundary are modeled using delegation connectors (ProvidedDelegationConnector and RequiredDelegationConnector).

An example of an assembled composite component was shown in Figure 2.2 in Chapter 2. Component CompA comprises three instances of basic components introduced in Figure 2.1 connected according to
Figure 4.3: Example of a nested composed structure showing the component types.

Figure 4.4: Example of a nested composed structure showing the composition of the components.

their provided and required interfaces.

Figure 4.3 and Figure 4.4 show an example of a nested composed structure illustrated with two different types of diagrams. The diagram in Figure 4.3 shows the involved component types and assembly contexts: System A consists of two assembly contexts referencing composite component X which itself consists of two assembly contexts that reference composite component Y. Composite component Y again consists of an assembly context referring to the basic component Z. The diagram in Figure 4.4 (composite diagram) depicts the composition of the involved composed structures. Note that the delegation connectors have been omitted in the diagram for reasons of clarity. As illustrated in Figure 4.3, there is one assembly context referring to the basic component Z, but as evident from Figure 4.4, there are four instances of component Z in System A. Thus, an assembly context is not equivalent to a component instance. An assembly context is only unambiguous within its direct parent composite structure.

Each component instance of Z can be uniquely identified by specifying the list of assembly contexts (i.e., the path) when navigating from the system boundary to the specific instance, e.g., X_AC1, Y_AC1, Z_AC1. This issue will be of relevance later when we consider performance-relevant behavior both at the component type level and at the component instance level.

4.1.2 Component Context

In order to capture the time behavior and resource consumption of a component’s provided service, PCM takes into account the four influencing factors shown in Figure 4.5. Obviously, the service im-
Motivation and Background

4.1.2.1 Component Type Level

In PCM, these four factors can be accounted for by specifying model variables in dependence on parameters that are available at the component boundary, i.e., service input parameters or return parameters of external service calls. Examples of influenced model variables are branching probabilities in the control flow of the service or resource demands of an internal action of the service. In PCM, parameter dependencies are specified at the component type level. See Figure 2.3 in Section 2.1.2 for an example.

However, we observed that in many practical situations, the behavior of software components is often influenced by parameters that are not available as input parameters passed upon service invocation. For instance, influencing parameters might be stored in a database structure queried by the invoked service. Given that it is unfeasible to model database states explicitly, such parameters are normally not modeled explicitly as part of performance models. This type of situation is typical for business information systems and our meta-model must provide means to deal with it.

4.1.2.2 Component Instance Level

In the context of DMM, model variables are typically characterized based on measurement data collected at run-time. The measurements are gathered at component instances. Thus, the question arises if the measurements, e.g., of a branching probability collected at an instance of a certain component type are representative for the corresponding branching behavior at another instance of the same component type.

Figures 4.6 and 4.7 underline the relevance of this question. Figure 4.6 shows an application server cluster for CustomerA, modeled as a Subsystem. The cluster consists of two instances of a composite component implementing the SPECjEnterprise2010 benchmark application. When a new node is added to the cluster, a new SPECjEnterprise2010 component instance is created. Intuitively, the characterizations of the model variables for the existing SPECjEnterprise2010 instances can be assumed to be representative also for the new SPECjEnterprise2010 component instance given that the different instances provide identical functionality and are associated with the same customer, i.e., they access the same underlying business data.

Figure 4.7 shows two subsystems, representing the application server clusters for CustomerA and CustomerB, respectively. In this case, it is not clear if the characterizations of model variables in the
SPECjEnterprise2010 component are representative across the boundaries of each subsystem. Although both CustomerA and CustomerB use the same component types, the underlying business data that the CRM, manufacturing and SCM applications of SPECjEnterprise2010 access is different for each customer. Thus, whether the characterization of a model variable is valid across subsystem boundaries depends on the specific variable considered. For the same component type, there can be model variables that are different for each component instance, or there might be model variables that can be treated as identical across all instances of the component type. Therefore, to deal with such situations, our metamodel should provide means to specify the scope of model parameters explicitly. We return to this issue in Section 4.1.4.

4.1.2.3 Deployment Level (Deployed Component Instance)

Besides model variables describing the control flow behavior, variables describing the resource demanding behavior of a service are of major importance in performance modeling. In an online scenario, the resource demands can be obtained from online measurements and hence they would be specific to the considered deployment environment. This is in contrast to PCM, where resource demands are generally intended to be specified in platform independent units.

Figure 4.8 describes a scenario where deployment-specific resource demands are required. The specification of component CompB contains a ResourceDemandingElement that is annotated with resource demands. Two instances of the component that are deployed on two different servers ApplicationServer1 and ApplicationServer2. Different resource demand specifications are specified for the two component instances. At ApplicationServer1, the resource demand amounts to 15ms CPU demand. At
Figure 4.8: Example: Deployment-specific resource demands.

ApplicationServer1, the resource demand is specified as 12ms CPU demand. Thus, to deal with such situations, our meta-model should provide means to specify resource demand in dependence on the deployment environment in which a given component is deployed. We refer to such resource demands as deployment specific resource demands.

### 4.1.3 Service Abstraction Levels

In the scenario described in Chapter 3, in order to ensure 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. This calls for online performance prediction capabilities. Some typical questions that arise during operation are: What performance would a new service deployed on the virtualized infrastructure exhibit and how much resources should be allocated to it? How should the system configuration be adapted to avoid performance issues or inefficient resource usage arising from changing customer workloads? What would be the effect of changing resource allocations and/or migrating a service from one physical server to another? 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. Thus, flexibility in trading-off between model accuracy and the overhead for building and solving the models is critically important. 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 maximum 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 speed.

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 the component behavior is available, depending on the platform on which the component is deployed, some parame-
ter dependencies 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 more appropriate to use a coarse-grained or black-box abstraction of the component behavior which only requires observing its behavior at the component boundaries.

In the following, we describe three practical examples in the context of our scenario where models at different abstraction levels would be needed. We consider the supplier domain of SPECjEnterprise2010 (see Section 3.2). Whenever the inventory of parts in the manufacturing domain is getting depleted, a request is sent to the supplier domain to order parts from suppliers. The supplier domain places a purchase order with a selected supplier offering the required parts at a reasonable price. Figure 4.9 shows the SupplyChainManagement (SCM) component providing a purchase service for ordering parts.

- If we imagine that the SCM component is an outsourced service hosted by a different service provider, the only type of monitoring data that would typically be available for the purchase service is response time data. In such a case, information about the 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 SCM 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 (using techniques such as, e.g., [42, 18]) as well as frequencies of calls to other components. Thus, in this case, a more fine granular model of the component can be built, allowing to predict its response time and resource utilization for different workloads.

- Finally, if the internal behavior of the SCM 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. Predictions of response time distributions are relevant for example when evaluating SLAs with service response time limits defined in terms of response time percentiles. In our scenario, as shown in Figure 4.10, the SCM component is implemented as a composite component containing a child component PurchaseOrder. The latter is responsible for dispatching the purchase orders (service sendPurchaseOrder). The sending operation supports two modes of operation: i) sending the order as an inline message without attachments, or ii) sending the order as a message with attachment. Figure 4.11 shows some measurements of the response time of sendPurchaseOrder as a histogram. The measurements were obtained during a benchmark run under medium load with a steady state time of 15 minutes. As expected, the measured response time distribution is multi-
modal. Thus, to predict the response time distribution of the sendPurchaseOrder operation, a fine-granular model of its internal behavior is needed taking into account its internal control flow and in particular the branch leading to either sending an inline message or sending a message with an attachment which in this case is performance relevant.

In summary, service behavior descriptions should be modeled 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 predictions. The modeled abstraction level depends on the information monitoring tools can obtain at run-time, e.g., to what extent component-internal information is available. Furthermore, different model solving strategies require different input data, e.g., fine-grained simulation models can deal with more detailed data than a single workload class queueing model.

### 4.1.4 Parameter Dependencies

Figure 4.12 shows the Manufacturing component of our SPECjEnterprise2010 scenario (see Chapter 3).\(^1\) The component provides a service scheduleManufacturing to schedule a new work order

---

\(^1\)Note that the names of some SPECjEnterprise2010 components and services have been adapted, i.e., renamed, in this technical report to ease understanding of their semantics.
in the manufacturing domain for producing a set of assemblies. The component is instantiated two times corresponding to two deployments of the SPECjEnterprise2010 application in our scenario, for Customer A and Customer B, respectively. A work order consists of a list of assemblies to be manufactured and is identified with a workOrderId. In case the items needed to produce the assemblies available in the manufacturing site’s warehouse are not enough, the purchase service of the SCM component is called to order additional items.

We are now interested in the probability of calling purchase which corresponds to a branch probability in the control flow of the scheduleManufacturing service. This probability will depend on the number of assemblies that have to be manufactured and the inventory of parts in the customer’s warehouse. The higher the number of assemblies, the higher the probability of having to purchase additional parts. Given that two different deployments of the application are involved, the respective probabilities for the two component instances of type Manufacturing can differ significantly. For instance, a customer with a large manufacturing site’s warehouse will order parts less frequently than a customer who orders items “just in time”.

As discussed in Section 2.1.2, PCM allows to model dependencies of the service behavior (including branch probabilities) on input parameters passed over the service’s interface upon invocation. However, in this case, the only parameter passed is workOrderId which refers to an internal structure stored in the database. Such a parameter does not allow to model the dependency without having to look into the database which is external to the modeled component. Modeling the state of the database is extremely complex and infeasible to consider as part of the performance model. This situation is typical for modern business information systems where the behavior of business components is often dependent on persistent data stored in a central database. The interface between component and database is too generic to infer direct parameter dependencies. Thus, in such a scenario, the PCM approach of providing explicit characterizations of parameter dependencies is not applicable.

To better understand the considered dependency, in Figure 4.13 we show that the Manufacturing component is actually triggered by a separate Dealer component providing a newOrder service which calls the scheduleManufacturing service. The newOrder service receives as input parameters an assemblyID and quantity indicating a number of assemblies that are ordered by a dealer. This information is stored in the database in a data structure (see Figure 4.14) using workOrderId as a reference which is then passed to service scheduleManufacturing as an input parameter.

Intuitively, one would assume the existence of the following parameter dependency: The more assemblies are ordered (parameter quantity of service newOrder of the Dealer component), the higher the probability that new items will have to be purchased to refill stock (i.e., probability of calling purchase...
in the Manufacturing component). However, this dependency cannot be modeled using the PCM approach since two separate components are involved and furthermore an explicit characterization is impractical to obtain.

In such a case, provided that we know about the existence of the parameter dependency, we can use monitoring statistics collected at run-time to characterize the dependency probabilistically. Figure 4.15 a) shows monitoring statistics that we measured at run-time showing the dependency between the influencing parameter quantity and the observed relative frequency of the purchase service calls. For instance, if the quantity equals to 20, in roughly one out of every four calls of scheduleManufacturing, service purchase was observed to be called. Figure 4.15 b) shows how the dependency can be characterized probabilistically by considering three ranges of possible quantities. For example, for quantities between 50 and 100, the probability of a purchase call is estimated to be 0.67.

Monitoring may introduce a non-negligible overhead. However, our experiments showed that it is possible to gather the relevant measurements without impacting the overall system performance significantly: The monitoring data was collected using the WebLogic Diagnostics Framework (WLDF) of Oracle WebLogic Server (WLS). We instrumented the service entries of scheduleManufacturing and the external service call purchase. Whenever an instrumented service is called, the monitoring framework creates an event record. Using the diagnostic context id provided by WLDF, the event records can be mapped to individual call paths, i.e., it is possible to map a service entry to an external service call. For a detailed explanation of how such call path traces can be extracted, see, e.g., [18]. We employed monitor throttling strategies provided by WLDF to control the number of requests that are processed by the instrumented monitors. Only a subset of the system requests have actually been tracked. That way, it is possible to obtain monitoring samples while keeping the monitoring overhead low. We conducted one benchmark run where we gathered the above statistics, and one benchmark run where we disabled the monitoring facilities. In both cases, the utilization of the application server CPU was about 40%. Thus, the average system utilization was not influenced significantly. Of course, the individual response time of a system request which is actually traced is affected. However, the sampling rate is configurable and
can be configured taking the trade-off between overhead and monitoring information into account.

We conclude that the behavior of software components (e.g., control flow and resource demands) is often dependent on parameters that are not available as input parameters passed upon service invocation. 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 type of situation is typical for business information systems and our meta-model must provide means to deal with it.

4.1.5 Virtualized Resource Landscape

The scenario presented previously in Chapter 3 is only a small example of how today’s 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. In 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 re-configuration decisions must cover diverse aspects and concepts. However, current performance models do usually not store 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 Chapter 5.2.

4.1.5.1 Resource Landscape

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 future work.

![Network Computing Storage](image)

Figure 4.16: 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.

4.1.5.2 Layers of Resources

A reoccurring pattern in modern virtualized data centers is the nested containment of system entities. For example, imagine data centers containing servers, which might contain a virtualization platform and virtual machines, again containing an operating system, containing a middleware layer and so on. This leads to a tree of nested entities. Because of this flexibility a large variety of different executing environments can be realized but all consist of similar, reoccurring elements. The information about how resource containers are stacked is also important for reconfiguration (e.g., to decide whether an entity can be migrated or not) but the more important fact is the influence 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.

4.1.5.3 Reuse of Modeled 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.

Therefore, to ease the modeling effort, it is necessary to have a concept which allows With the meta-model concepts presented so far, 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 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 Concepts

We present a novel performance meta-model specifically designed for building online architecture-level performance models that enable performance prediction online during system operation.

We use PCM [1] as a basis, given that it provides support for explicitly modeling the performance-influencing factors of services, i.e., their software architecture (components, control flow, resource demands), execution environment (resources, configuration parameters), deployment (resource allocation)
and usage profile (number of users, service request rates, input parameters) [16]. For a discussion of the differences between PCM and DMM refer to Chapter ??.

The remainder of this section is structured as follows: In Section 4.2.1 we describe how we model the service behavior in DMM. The modeling approach for parameter and context dependencies is presented in Section 4.2.2, the resource landscape meta-model is shown in Section 4.2.3.

4.2.1 Service Behavior Abstractions for Different Levels of Granularity

As motivated in Section 4.1.3, service behavior descriptions should be modeled 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 predictions. Moreover, the modeled abstraction level depends on the information monitoring tools can obtain at run-time, e.g., to what extent component-internal information is available.

**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 demanding behavior is 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 behavior.

- **Coarse-grained behavior abstraction.** A “coarse-grained” abstraction captures the component behavior when observed from the outside at the component 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 is similar to the RDSEFF in PCM. The control flow is modeled at the same abstraction level as in PCM, however, our approach has some significant differences in the way model variables and parameter dependencies are modeled. The details of these are presented in detail in Section 4.2.2. 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.

  Note that it is not required that a course-grained behavior model of a service is fully consistent with a fine-grained behavior abstraction of that service. Automatic transformations from a fine-grained behavior model to a course-grained model are yet out of scope.

**Meta-model** Figure 4.18 shows the meta-model elements describing the three proposed service behavior abstractions. Type FineGrainedBehavior is attached to the type BasicComponent, a component type that does not allow containing further subcomponents. The CoarseGrainedBehavior is attached to the type InterfaceProvidingRequiringEntity that generalizes the types System, Subsystem, CompositeComponent and BasicComponent. Type BlackBoxBehavior is attached to type InterfaceProvidingEntity, neglecting external service calls to required services. Thus, in contrast to the fine-grained abstraction level, the coarse-grained and black-box behavior descriptions can also be attached to service-providing *composites*, i.e., ComposedStructures. Note that the type hierarchy for entities and components stems from PCM.
**Figure 4.18: Service behavior abstraction.**

**Figure 4.19: Black-box behavior (a) and coarse-grained abstractions (b).**

**Figure 4.20: Fine-grained behavior abstraction.**
The meta-model elements for the CoarseGrainedBehavior and BlackBoxBehavior abstractions are shown in Figure 4.19. A CoarseGrainedBehavior consists of ExternalCallFrequencies and ResourceDemandSpecifications. An ExternalCallFrequency characterizes the type and the number of external service calls. Type ResourceDemandSpecification captures the total service time required from a given ProcessingResourceType. A BlackBoxBehavior, on the other hand, can be described with a ResponseTime characterization.

Figure 4.20 shows the meta-model elements for the fine-grained behavior abstraction. A Component-InternalBehavior 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. 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; otherwise, the whole loop would be captured as part of an InternalAction). LoopActions and BranchActions can be characterized with loop iteration counts and branch probabilities, respectively.

4.2.2 Parameter and Context Dependencies

In Section 4.1.4 we observed that the behavior of software components is often dependent on parameters that are not available as input parameters passed upon service invocation (i.e., “hidden” parameters). 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 type of situation is typical for business information systems and our meta-model must provide means to deal with it.

Modeling Approach. To allow the modeling of the above-mentioned “hidden” parameter dependencies, in addition to normal call parameters, our performance meta-model supports the definition of arbitrary influencing parameters where call parameters are treated as a special case. In order to resolve parameter dependencies, the influencing parameters need to be mapped to some observable parameters that would be accessible at run-time and traceable to the system boundary. Often such a mapping will only be feasible at deployment time once the complete system architecture and execution environment is available.

Figure 4.21 illustrates our modeling approach in the context of the presented example from Figure 4.13. The branch probability of calling the purchase service within the control flow of the service scheduleManufacturing is represented as InfluencedVariable1. The component developer is aware of the existence of the dependency between the branch probability and the quantity of assemblies to be manufactured. However, he does not have direct access to the quantity parameter 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 defines an InfluencingParameter1 representing the “hidden” quantity parameter and provides a semantic description as part of the component’s performance model. He can then declare a dependency relationship between InfluencedVariable1 and InfluencingParameter1.
The developer of composite component SPECjEnterprise2010 is then later able to link InfluencingParameter1 to the respective service call parameter of the Dealer component, designated as InfluencingParameter2. We refer to such a link as declaration of a correlation relationship between two influencing parameters. In our example, the correlation can be described by the identity function. Having specified the influenced variable and the influencing parameters, as well as the respective dependency and correlation relationships, the parameter dependency then can be characterized empirically as discussed earlier (Figure 4.15). Our modeling approach supports both empirical and explicit characterizations for both dependency and correlation 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 be considered as auxiliary model entity allowing to model parameter dependencies in a more flexible way. 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.

Finally, the same software component might be used multiple times in different settings, e.g., as in our scenario where the same application is run on behalf of different customers in separate virtual machines with customized application components. Hence, the meta-model should provide means to specify the scope of influencing parameters. A scope of an influencing parameter specifies a context where the influencing parameter is unique. This means, on the one hand, that measurements of the influencing parameter can be used interchangeably among component instances provided that these instances belong to the same context. On the other hand, it means that measurements of the influencing parameter are not transferable across scope boundaries. Thus, if monitoring data for a given influencing parameter is available, it should be clear based on its scope for which other instances of the component this data can be reused.
Information about parameter scopes is particularly important when using online performance models to predict the impact of dynamic reconfiguration scenarios. For instance, when considering the effect of adding server nodes to the application server cluster of a given customer (hosting instances of our SPECjEnterprise2010 composite component), given that influencing parameters within the cluster belong to the same context, monitoring statistics from existing instances of the SPECjEnterprise2010 component can be used to parameterize the newly deployed instances.

The concepts of scope, dependency and correlation relationships can be combined. In the following, we present the meta-modeling facilities for influenced variables, influenced parameters, dependencies, correlations and scopes in more detail.

**Meta-Model** We first introduce the meta-model elements for the model variables. Then, we provide the meta-model of the relationships and their scope-dependent characterization.

**Model Variables.** The model variables involved in dependency specifications are divided into influenced variables and influencing parameters. As shown in Figure 4.22, model variables that can be referenced as InfluencedVariable include resource demands and control flow variables such as branch probabilities, loop iteration count distributions and call frequencies (for coarse- and fine-grained behavior descriptions) and response times (for black box behavior descriptions). Parameters having an influence on the model variables are represented using the entity InfluencingParameter. Normal service call parameters such as service input parameters, external call parameters or return parameters of external calls (see Figure 4.23) are special types of influencing parameters.

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, following the characterizations of variables as they are used in PCM, our meta-model supports referring not only to a parameter VALUE, but also to other characterizations such as NUMBER_OF_ELEMENTS if the referred parameter is a collection.

An InfluencingParameter is attached to a service behavior abstraction and has a designated name and description. These attributes are intended to provide a human-understandable semantics that could be used by component developers, system architects, system deployers or run-time administrators to identify and model relationships between the model variables.

**Relationships: Dependency and Correlation.** As shown in Figure 4.25, we distinguish the two
Figure 4.23: Call parameter hierarchy.

Figure 4.24: Call parameters.
types of relationships DependencyRelationship and CorrelationRelationship between model variables. The former declares an influenced variable to be dependent on an influencing parameter. The latter connects two influencing parameters declaring the existence of a correlation between them. The Relationship entities are attached to the innermost (composite) component or (sub-)system that directly surrounds the relationship. A dependency is defined at the type-level of the component and is specified by the component developer. In this report, for reasons of clarity, we only consider one-dimensional dependencies. In general, our meta-model supports the modeling of multi-dimensional dependencies where influenced variables are dependent on multiple influencing parameters.

A correlation is specified when a composed entity such as a Sub-system is composed of several assembly contexts. Thus, both sides of the correlation, designated as “left” and “right”, are identified not only by an InfluencingParameter but also by the specific component instance where the influencing parameter resides.

To provide maximum flexibility, it is possible to map the same InfluencingParameter to multiple co-related InfluencingParameters, some of which might not be monitorable in the execution environment, others might 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 that are usable might involve different monitoring overhead. Given that the same mapping might 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.

Finally, note that an AssemblyContext cannot always serve as unique identifier of a component instance. For example, imagine a subsystem containing several instances of the SPECjEnterprise2010 component of Figure 4.21 representing a customer-specific application server cluster. From the subsystem’s perspective, the different component instances of, e.g., the Manufacturing component, cannot be distinguished by one AssemblyContext since this context is the same among all instances of the SPECjEnterprise2010 component. Hence, in order to unambiguously identify a certain Manufacturing instance from the perspective of such a customer-specific subsystem, we require the specification of a path consisting of the AssemblyContext of the SPECjEnterprise2010 component followed by the AssemblyContext of the Manufacturing component. Accordingly, in our meta-model, such paths
The default case is when an influencing parameter is globally unique (at the component type level). In this case, monitoring data from all observed instances of the component can be used interchangeably and treated as a whole. Moreover, once a declared dependency of the component behavior on this influencing 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. This trivial case can be modeled by either omitting the specification of scopes or by specifying a scope referencing the global system.

**Characterization of Relationships.** Each dependency or correlation relationship can be characterized using either an ExplicitCharacterization or an EmpiricalCharacterization (see Figure 4.26). The former means that the relationship is characterized using explicit parameter dependencies as known from PCM. Such a characterization is suitable if an expression of the functional relation between the two considered model variables is available. If this is not the case, an EmpiricalCharacterization can be used to quantify the relationship using monitoring statistics. The entity EmpiricalFunction in the figure describes the interface to a characterization function based on empirical data. An EmpiricalFunction can for example be based on monitoring statistics as discussed earlier (Figure 4.15 a) in Section 4.1.4) and can be represented by summarizing the obtained statistics as illustrated in Figure 4.15 b).

Besides using, e.g., equidistant or equiprobable bins, more complex approaches to calculate representative histograms can be found in the area of query optimization for databases[43]. In that area, multidimensional histograms are used to model the distribution of relational data. Static histograms, e.g., MHIST [44] or GENHIST [45], are constructed based on the underlying dataset before they can be utilized. Dynamic histograms like STHoles [46] or ISOMER [47] are constructed on-the-fly: Instead of having to analyze the complete dataset at the time of construction, dynamic histograms are built using the results of queries. Dynamic histograms are often also called self-tuning histograms as they do not require constant histogram rebuilds [48, 46]. The latter property makes them appealing for the usage in our context.

In this technical report, we focus on the modeling concepts at the meta-model level. We omit descriptions of how the empirical characterizations can be derived from monitoring statistics. In the following, we formalize the semantics of the characterization functions for dependency relationships and correlation relationships.

Let \( v \) be an influenced variable of a dependency relationship and \( p \) the respective influencing parameter. A corresponding explicit function \( f \) then has the signature \( f : \text{Literal} \rightarrow \text{ProbabilityFunction} \). It maps a value of \( p \) to a probability function describing \( v \). The signature of an empirical function \( \hat{f} \) is \( \hat{f} : \text{Literal} \times \text{AssemblyContext} \rightarrow \text{ProbabilityFunction} \). This function also maps a value of \( p \) to a probability function. However, depending on the defined scope \( S(p) \), the function has to evaluate differently for different component instances of \( C(p) \). We denote a component instance as \( \text{inst}_{C(p)} \).

According to the function’s signature, \( \hat{f} \) evaluates depending on an assembly context. More precisely, \( \hat{f} \) evaluates depending on the assembly context of the scope-specifying composite of \( \text{inst}_{C(p)} \), i.e., the
The formalization of the semantics of characterization functions for correlation relationships is similar. The main difference is that a correlation relationship may provide two functions for both directions “left to right” and “right to left”, i.e., $f_{\text{leftToRight}}$ and $f_{\text{rightToLeft}}$ for explicit functions, respectively $\hat{f}_{\text{leftToRight}}$ and $\hat{f}_{\text{rightToLeft}}$ for empirical functions. For explicit functions $f_{\text{leftToRight}}$ and $f_{\text{rightToLeft}}$, the scopes of the involved influencing parameters $p_{\text{left}}$ and $p_{\text{right}}$ are ignored. For the empirical functions $\hat{f}_{\text{leftToRight}}$ and $\hat{f}_{\text{rightToLeft}}$, the intersection of the scopes $p_{\text{left}}$ and $p_{\text{right}}$ has to be considered. An important special case concerning the correlation relationship between two InfluencingParameters is the parameter propagation. In that case, the correlation from “left to right” and vice versa corresponds to the identity function.

4.2.3 Dynamic Virtualized Resource Landscapes

In the following section we present a meta-model to model 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 presented modeling approach is generic covering all types of resources. However, in this version of the technical report we focus on the computational infrastructure. Modeling other aspects like storage or network infrastructure with switches or storage services are briefly considered, but they are part of future work. In Section 5.2.1, we introduce the meta-model to annotate this static view of the resource landscape with the dynamic aspects, i.e., the reconfigurable parts of the system.

4.2.3.1 Resource Landscape

Figure 4.27 depicts an high-level view and the overall structure of the resource landscape meta-model as an UML class diagram. The root entity in the model that comprises all other model elements is the DistributedDataCenter. A DistributedDataCenter consists of one or more DataCenters.
DataCenters contain AbstractHardwareInfrastructures, i.e. CompositeHardwareInfrastructures or the three types ComputingInfrastructure, NetworkInfrastructure, and StorageInfrastructure. A CompositeHardwareInfrastructure is a structuring element to group AbstractHardwareInfrastructures, e.g., in server racks or clusters. Current architecture-level performance models usually abstract from these details and do not reflect the hierarchy of resource containers. However, for resource management at run-time, the resource landscape and hierarchy of resources is crucial to make better decisions, e.g., to decide if a VM can be migrated and where it should be migrated to. The ComputingInfrastructure models the actual physical (computational) machines containing and executing different layers of software elements. The relationship of these contained elements is explained in the following Section 4.2.3.1. Meta-modeling StorageInfrastructure and NetworkInfrastructure in more detail is out of scope for this version of the technical report and part of future work.

Containers and Containment Relationships To model the hierarchy of nested system entities and resource layers, we introduce containers and a containment relationship. The central entity of the resource landscape meta-model depicted in Figure 4.27 is the abstract entity Container. The Container has a property configSpec to specify its configuration (which will be explained in Section 4.2.3.1) and a property template referring to a ContainerTemplate (which will be explained in Section 4.2.3.1). Most important is that a Container can contain further containers to model the tree-like structure of entities. In our meta-model, we distinguish between two major types of containers, the ComputingInfrastructure and the RuntimeEnvironment. The ComputingInfrastructure forms the root element in our tree of containers and corresponds to the physical machines of data centers. It cannot be contained by another container but it can have nested containers. The RuntimeEnvironment is a generic model element to build nested system layers, i.e., it can be contained within a container and it might contain further containers. Each RuntimeEnvironment has the property ofClass to specify the class of the RuntimeEnvironment. These classes are introduced in the following Section 4.2.3.1.

Figure 4.27: Resource landscape meta-model.
Classes of Runtime Environments  We distinguish six general classes of runtime environments which are listed in Figure 4.28. These are HYPERVISOR for the different hypervisors of virtualization platforms, OS for operating systems, OS_VM for virtual machines emulating standard hardware, PROCESS_VM for virtual machines like the Java VM, MIDDLEWARE for middleware environments, and OTHER for any other type. By setting the ofClass property of the RuntimeEnvironment to one of these values, it is possible to enforce consistency within the modeled layers, e.g., by using OCL constraints in the meta-model. The constraint we implemented prohibits the instantiation of different RuntimeEnvironment classes within one container:

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

![Enumeration of RuntimeEnvironmentClasses](image)

Figure 4.28: Different runtime environment classes.

As a result, a RuntimeEnvironment can only contain containers of the same class, e.g., a hypervisor can only contain virtual machines.

We could have introduced the different types of runtime environments as explicit model entities but we focused on designing a model supporting extendability. Modeling all classes of runtime environments as explicit model entities would have required to also model their relations (e.g., OS_VM can only be contained in HYPERVISOR etc.) which renders the 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’s structure making it easier to reuse and extend the model instances at run-time. Of course, not modeling the relations between the classes has the disadvantage of building model instances which are wrong. However, we assume that models instances can be built automatically and the tool support is aware of such constraints (e.g., by OCL constraints as the previous one).

**Example 4.1.** Figure 4.29 depicts an example model instance with different container types. It models a virtualized distributed data center DDC consisting of one data center DC. Although in this example the distributed data center contains only one data center, we must model this entity because it is the root element of the meta-model. Data center DC contains a Rack which groups the computing infrastructure Server and Network and Storage. Server contains a virtualization platform XenServer5.5. More specifically, this container is a RuntimeEnvironment of class HYPERVISOR running two VMs, VM1 and VM2. Also, VM2 contains a middleware, modeled by the runtime environment WebLogic of class MIDDLEWARE. This example demonstrates the principle of stacking several resource layers with the containment relation. It shows that it is possible to model very complex resource landscape scenarios. In the next step we have to specify the physical and logical resources that are actually provided by the containers of the computing infrastructure.
Resource Configuration Each Container can have its own specific resource configuration. We distinguish between three different types of configuration specifications: ActiveResourceSpecification, PassiveResourceSpecification, and CustomConfigurationSpecification (see Figure 4.30(a)).

The purpose of the ActiveResourceSpecification is to specify the active resources a Container provides. An example is the CPU as the ActiveResourceSpecification of a physical or virtual machine. One can use the ProcessingResourceSpecification and/or LinkingResourceSpecification to specify what ProcessingResourceTypes and CommunicationLinkResourceTypes the modeled entity offers. The currently supported ProcessingResourceTypes are CPU and HDD, and LAN for
the CommunicationLinkResourceType, which are stored in a resource type repository. The ProcessingResourceSpecification is further defined by its properties schedulingPolicy and processingRate. For example, a CPU would be specified with PROCESSOR_SHARING as schedulingPolicy and a processingRate of 2.4 GHz. If a ProcessingResourceSpecification has more than one processing units (e.g. a CPU has four cores), the attribute number of NumberOfParallelProcessingUnits would be set to 4, whereas two CPUs can be modeled as two separate ProcessingResourceSpecifications. The LinkingResourceSpecification specifies a bandwidth and can be used to model a network interface card.

The PassiveResourceSpecification can be used to specify properties of passive resources. Passive resources can be, e.g., the main memory size, database connections, the heap size of a JVM, or resources in software, e.g., thread pools. Passive resources refer to a PassiveResourceCapacity, the parameter to specify, e.g., the number of threads or memory size.

In case a Container has a very individual configuration which cannot be modeled with the previously introduced elements, one can use the CustomConfigurationSpecification. This configuration refers to the EMF element EObject, i.e., one can refer to any custom EMF model reflecting the relevant configuration of this container. For example, imagine the configuration of a hypervisor with all its properties that influence the performance of virtual machines. In [19], we categorized the most performance-relevant aspects of the two major hypervisor architectures and modeled their impact on the performance overhead of virtualization. With the CustomConfigurationSpecification we enable the possibility to refer to such a model.

**Container Types** This concept enables the differentiation between container types and instances of these types. A container type specifies the general performance properties relevant for all instances of this type and the instances store the performance properties of each container instance.

With our solution one can specify ContainerTemplates and collect them in the ContainerRepository (see Figure 4.30(b)). The ContainerTemplate is similar to a Container because it also includes a ConfigurationSpecification which specifies the configuration of the ContainerTemplate. A Container in the resource landscape model might have a reference to a ContainerTemplate (see Figure 4.27). The advantage of this template mechanism is that the general properties relevant for all instances of one container type can be stored in the container template and the relevant configuration specific for an individual container instance can differ. When analyzing the model, the specific individual properties override the settings of the template. For example, assume that a container instance has no individual properties and only a reference to a template. Then, only the configuration specification of the template would be considered. However, if the container instance has an individual configuration specification, then these settings would override the properties of the template.

Another solution would have been to develop a second meta-model for the general properties of a container, i.e., to model the container types. This meta-model would act as a “decorator model”, i.e., it would extend a resource landscape model instance. In the next step, one could then instantiate the types created in this decorator model. The drawback of this solution is that this would introduce a further level of meta-modeling, i.e. and additional meta-model for container types to create type instances. However, this would require that a container provider (e.g. a virtualization platform vendor) must be familiar with meta-modeling.

**Example 4.2.** Figure 4.31 depicts an example model instance of the container template repository. In this example, we model i) a XenServer5.5 hypervisor, ii) a virtual machine with a virtual CPU with eight cores, and iii) a middleware container with a thread pool.
Accordingly, the model instance of the template repository contains three container templates: XenHypervisor, VM1, and WebLogic, all templates of type RuntimeEnvironment.

The XenHypervisor can serve as template for containers which run the Xen hypervisor. It has a CustomConfigurationSpecification referencing the XenModel which is an instance of an exemplary HypervisorPerformanceModel. This template can be referenced, e.g., by the XenServer5.5 container in Example 4.1.

The VM1 template contains one ActiveResourceSpecification describing the virtual CPU of the VM. In this case, the virtual CPU of VM1Config is a ProcessingResourceSpecification referring to the resource type CPU with the processingRate of 2.4GHz and eight cores as NumberOfParallelProcessingUnits.


4.2.3.2 Deployment Model

After describing the resource landscape, one must specify which services are executed on which part of the resource landscape. We refer to this specification as deployment captured in the deployment model depicted in Figure 4.32. The deployment model is based on the Palladio Component Model (PCM) which also models the allocation of software components to resource containers in a separate allocation context model [49]. 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.

Our deployment model associates a service assembly (named AssemblyContext) with a container instance of the resource landscape model. A Deployment has a reference to a DistributedDataCenter, i.e., a resource landscape model instance. More importantly, the Deployment contains several DeploymentContexts. The DeploymentContext is the mapping of an AssemblyContext to a Container. An AssemblyContext stores the information how instances of services are assembled. For example, an AssemblyContext of ServiceA keeps the information to which other services ServiceA is connected. Furthermore, the AssemblyContext enables to distinguish between instances of a service, i.e., different instances of the same service can then be deployed on different containers. Thereby it is possible to
model redundant deployment of services on different machines or to create instances of the same service but with different QoS, because they are deployed on different containers. The Deployment has a reference to the System because AssemblyContexts are stored in the System.

![Deployment Model Diagram]

Figure 4.32: The deployment model.

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 [17]. 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.
Chapter 5

Model-based System Adaptation

In this chapter, we introduce the parts of the Descartes Meta-Model relevant to i) describe the dynamic aspects of modern IT systems, infrastructures and services (the adaptation points model) and to ii) model autonomic resource management at run-time (the S/T/A adaptation language). 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 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, the Descartes Meta-Model 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 Descartes Meta-Model should also contain means to describe these system adaptation processes and heuristics, abstracting from the system specific details. In summary, DMM 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.

We explicitly separate the Descartes Meta-Model 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. It comprises application architecture and resource landscape (see Figure 1.2), i.e., the parts of the Descartes Meta-Model introduced in Chapter 4.

Basically, the model contains 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. Hence, other architecture-level performance models could be used, too, e.g., PCM. For more details, about the system architecture sub-meta-model of DMM we refer to Chapter 4. The following sections motivate adaptation points model and 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. 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 captured by the model concepts. The modeling concept introduced in Chapter 4 are focused on the static aspects of the system. The adaptation points sub-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 [41] 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 Descartes Meta-Model will be introduced in Section 5.2.1.

**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 adap-
tation processes, e.g., rule-based system reconfiguration, or complex heuristics which adapt the system according to user-specific strategies [32]. 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 [33, 32]. 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 [33]). 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 Descartes Meta-Model are explained in Section 5.2.2.

5.2 Concepts

This section introduces the concepts of the adaptation points model (Section 5.2.1) and the S/T/A adaptation language (Section 5.2.2), which refers to the adaptation points model.

5.2.1 Adaptation Points Model

So far, the Descartes Meta-Model was focused on the static aspects of the system. However, modern virtualized data centers are increasingly dynamic and offer high flexibility for reconfiguring systems at run-time. In the following, we introduce an additional meta-model which addresses this flexibility and variability.

The Adaptation Points meta-model is an addition to the static system architecture QoS model introduced in Chapter 4. It can be used to annotate the static system architecture models at the instance level. For example, the annotations describe which parts of the resource landscape model are variable and can be reconfigured during operation, i.e., it provides possibilities to specify the configuration range of the dynamic system. However, it is not intended to specify how to change the model instance or even the system, i.e., the actual change itself is implemented in the adaptation process that uses the adaptation points model. Figure 5.2 depicts the overview of the adaptation points meta-model.

![Figure 5.2: Adaptation points model.](attachment:image.png)

The specifications, i.e., the annotations of the adaptable elements of a static model instance are col-
lected in the AdaptationPointDescriptions. We distinguish between two types of elements which may vary at run-time: a) model variables like PassiveResourceCapacity and b) other model entities, e.g., the number of instances of a model entity. These two types of AdaptationPoints are modeled as ModelVariableConfigurationRange (a) and ModelEntityConfigurationRange (b), respectively, which specify the range in which the static model instance can be varied.

One can now annotate the static model using these two adaptation points. The ModelVariableConfigurationRange refers to a AdaptableEntity and specifies the range in which the model variable can be changed at run-time using the attributes minValue and maxValue. AdaptableEntity is a class in our meta-model and all meta-model entities which are variable at run-time inherit from this type. Hence, at the meta-model level, all entities of type AdaptableEntity are considered as variable elements. However, on the model instance level, they are only considered to be variable if they are actually annotated by a AdaptationPoint. The reason is that even if a system has adaptation points there might be systems where it is prohibited to change these configuration. For example, a virtualized environment might prohibit changing the number of virtual CPUs for reliability reasons.

The meta-model entity ModelEntityConfigurationRange is used to annotate other static model instance entities that cannot inherit from AdaptableEntity, e.g., the instances of one specific container type. To this end, the ModelEntityConfigurationRange refers to an EObject and to a VariationType. The EObject can be any entity of, e.g., the resource landscape model instance, e.g., a Container. The VariationType specifies in more detail how this model entity can vary. Currently, we distinguish two variation types: the PropertyRange or the SetOfConfigurations. The idea of the PropertyRange is to specify a variability range using two OCL constraints (minValueConstraint and maxValueConstraint). They are also be used to check whether the variation is within the valid value range or not. An example would be 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 container on other containers. In this case, possible variants are references to other model instance elements (for example the different servers a VM can be deployed on) and are collected in the SetOfConfigurations.

In summary, this meta-model concentrates on the description of all possible states one single instance of a dynamic system might have. It is not intended to describe all possible instance variants a dynamic system might take.

Example 5.1. Figure 5.3 depicts an adaptation points model instance. It describes three relevant scenarios: i) adding and removing virtual CPUs to VMs, ii) adding and removing additional VMs, and iii) migrating VMs.

![Diagram](image-url)
To model an adaptable amount of virtual CPUs we use \texttt{VariableVcpu}, an instance of \texttt{ModelVariableConfigurationRange} which refers to \texttt{cores}, the \texttt{NumberOfParallelProcessingUnits} instance of \texttt{VM1} in Example 4.2. The attributes are set to \texttt{minValue = 2} and \texttt{maxValue = 8}, respectively. The result is that this parameter of the \texttt{ProcessingResourceSpecification} referencing this model entity can be varied between the values 2 and 8.

To model the adaptability of the number of VM instances, the \texttt{VariableVmInstances} references the container instance which should be variable. In this example, we refer to \texttt{VM1} template from Example 4.1. The property range of this model element is modeled by \texttt{InstanceRange}, specifying two OCL constraints to determine the property range. These OCL constraints are:

\begin{verbatim}
context ModelEntityConfigurationRange
inv minInstances:
   RuntimeEnvironment.allInstances()
   ->select (r | r.template = self.entity)
   ->size () > 1;
context ModelEntityConfigurationRange
inv maxInstances:
   RuntimeEnvironment.allInstances()
   ->select (r | r.template = self.entity)
   ->size () <= 8;
\end{verbatim}

and they ensure that there are at least two and at most eight instances of VMs which have the same template as the VM the property range is specified for.

The third configuration point is the \texttt{SetOfConfigurations} instance \texttt{VmMigration}. It is an example of how to specify that a VM can be migrated to other virtualized platforms. The \texttt{ModelEntityConfigurationRange} instance \texttt{VmMigration} refers to \texttt{VM2} from Example 4.1. Please note that although the actual location information of \texttt{VM2} is given by the reference \texttt{containedIn}, we cannot refer to this model element directly. The main problem of a direct reference would be that all \texttt{containedIn} references are variable. Hence, we must specify this additional information in the OCL constraint \texttt{migrationPossibilities} which restricts the migration variability of \texttt{VM2}. Again, this model instance is not intended to specify how the migration is actually done. The intention of this model is to specify the borders in which the configuration of a dynamic system can vary. The specification of the actual implementation of the configuration is part of the reconfiguration process.

\begin{verbatim}
context ModelEntityConfigurationRange::
migrationTargets () : Set {RuntimeEnvironment}
body: RuntimeEnvironment.allInstances
   ->select (r | r.ofClass = 'HYPERVISOR')
\end{verbatim}

5.2.2 Adaptation Language

In [20], we presented an algorithm for dynamic resource allocation in virtualized environments. This algorithm is currently implemented in Java. It is highly system-specific, unintuitive and difficult to maintain and reuse. We use this algorithm as a running example to illustrate the concepts of our adaptation language.

**Dynamic Resource Allocation Algorithm** The algorithm uses an system architecture QoS model we already developed and successfully applied for finding a system configuration that maintains given Service Level Agreements (SLA) while using as little resources as possible. The algorithm consists of two phases, a PUSH and a PULL phase. The PUSH phase is triggered by SLA violations observed in
the real system. The PULL phase is either triggered after the PUSH phase or by a scheduled resource optimization event. In the first step of the PUSH phase, the algorithm uses the system architecture sub-meta-model to estimate how much additional resources are needed to maintain SLAs, based on the current system state. Then, it increases the resources in the model up to this estimation. These steps are repeated until the predicted QoS fulfills the SLAs. Resources can be increased by either adding a virtual CPU (vCPU) to a virtual machine (VM) – in case additional cores are available – or by starting additional VMs running application servers and adding them to the application server cluster.

In the PULL phase, the algorithm uses the system architecture QoS model to estimate how much resources can be released without breaking SLAs. The amount of allocated resources are reduced stepwise by removing vCPUs from VMs and removing whole VMs from the application server cluster when their number of allocated vCPUs reaches zero. At each step, the system architecture QoS model is used to predict the effect of the system adaptation. If an SLA violation is predicted, the previous adaptation step is undone and the algorithm terminates. After the algorithm terminates successfully, the operations executed on the model are replayed on the real system. A more detailed description of this algorithm and its execution environment is given in [20].

Our S/T/A adaptation language consists of three major interacting concepts: **Strategy**, **Tactic**, and **Action**. Each concept resides on a different level of abstraction of the adaptation steps depicted in Figure 5.4. At the top level are the strategies where each strategy aims to achieve a given high-level objective. A strategy uses one or more tactics to achieve its objective. Tactics execute actions, which implement the actual adaptation operations on the system model or on the real system, respectively.

In this work, we use the terms strategy, tactic, and action as follows. A strategy captures the logical
aspect of planning a reconfiguration or adaptation as introduced previously. A strategy defines the objective that needs to be accomplished and conveys an idea for achieving it. 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 applied tactics is unknown, allowing for flexibility to react in unforeseen situations. For example, a defensive strategy in the PUSH phase of our example 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".

Tactics are the essential part of strategies. They are the technical aspect that follows the planning. Tactics are the part that actually execute the adaptation actions. Therefore, tactics specifically refer to actions. In the strategy phase of a plan, one thinks about how to act, i.e., one decides what tactics can be employed to fulfill the strategy’s objective depending on the current system state. However, in contrast to strategies, tactics specify precisely which actions to take without explicitly considering their effect which is done at the strategy level. A possible tactic of adding resources in our example could be "if possible, add another vCPU to a VM, otherwise, request another application server". More detailed examples are given in the following sections. Each tactic forms an encapsulated step in the adaptation process that is applied with atomic transactional semantics. After executing a tactic, we evaluate its impact using the system architecture QoS Model and then we decide what to do next.

The distinction of these three abstraction levels can be found in other approaches too, e.g., in [34] or [35], however, with limited expressiveness (cf. Section 2.2). In contrast to existing approaches, we propose a generic meta-model explicitly defining a set of modeling abstractions to describe strategies, tactics and actions with the full spectrum of self-adaptive mechanisms, from conditional expressions to complex algorithms and heuristics, 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 the proposed meta-model bottom up as depicted in Figure 5.5. We now explain the concepts depicted in in more detail and starting at the bottom with actions.

![Figure 5.5: Adaptation language meta-model.](image-url)
5.2.2.1 Action

Actions are the atomic elements on the lowest level of the adaptation language’s hierarchy. They execute an adaptation operation on the model or the real system, respectively. Actions can refer to Parameter entities specifying 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 use return values of executed actions as arguments for subsequent actions. In our meta-model, actions do not specify how the operation is actually implemented, neither at the model nor at the system level. Actions only refer to the corresponding points in the adaptation points model or in the system reconfiguration API, respectively. It is the responsibility of the framework interpreting the S/T/A adaptation language or of the system reconfiguration module, respectively, to provide the actual implementation.

Example

Figure 5.6 shows the four actions we modeled in our dynamic resource allocation algorithm. The actions addVCPU and addAppServer increase the resources used by the system, either by adding a vCPU to a VM (addVCPU) or by adding a new VM running an application server to the application server cluster (addAppServer). Similarly, removeVCPU and removeAppServer can be used to remove resources. These actions do not implement the logic of the operation, they are simply references to variation points defined in the respective system architecture QoS model, i.e., DMM instance in this case. On the model level, actions are implemented by the framework using the models. On the system level, the virtualization layer executes the respective operations on the real system.

5.2.2.2 Tactic

A Tactic specifies a AdaptationPlan with the purpose to adapt the system in a specific direction, e.g., to scale-up resources. The AdaptationPlan describes how the tactic pursues this purpose, i.e., in which order it applies actions to adapt the system. More specifically, each AdaptationPlan contains a set of AbstractControlFlowElements. The order of these control flow elements is determined by their predecessor/successor relations.

Concrete implementations of the AbstractControlFlowElement are Start and Stop as well as Loop and Branch. The purpose of these abstract control flow elements is to describe the control flow of the adaptation plan. For example, each Branch has an attribute condition which contains a condition directly influencing the control flow, e.g., by evaluating monitoring data, the system/model state or OCL expressions. Tactics can refer to Parameter entities 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.

An important property of tactics is their transaction-like semantic. We define tactics as: i) atomic, i.e., either the whole tactic with all its contained actions is executed or the tactic must be rolled back, ii) consistent, i.e., the model’s and system’s state must be consistent after applying a tactic, and
deterministic, i.e., tactics have the same output if applied on the same system state. This transac-
tion-like behavior is important because after applying a tactic at the model level, the effect of the performed
adaptation is evaluated by analyzing the QoS model, i.e., several actions can be executed at once without
having to analyze the model after each action. This can save costly model analysis time which is crucial
at run-time. Furthermore, applying tactics at the model level before applying them to the real system has
the advantage that we can test their effect when applied as a whole without actually changing the system.
Thereby, it is always possible to jump back to the state before starting to apply the tactic in case an error
is detected saving costly executions of roll-back operations on the system.

Example In Figure 5.7, we show the three tactics a system designer has specified for the example
based on the previously presented actions. These tactics are addResources, removeResources, and undoPreviousAction. The first two tactics are used to scale the system up or down, the third tactic
can be applied to undo a previous action.

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 should be added by executing the tactic. The adaptation plan of the tactic
chooses which resource type to add. This is an example for separating technical from logical details. The
adaptation plan in the Loop action implements two actions, addVCPU and addAppServer. Which action
is executed depends on the current system configuration. If there is no possibility to add another vCPU
determined using an OCL expression AllServersAtMaxCap), another application server VM is added.

The adaptation plan of the tactic removeResources either removes an application server VM if there
is a server running at minimum capacity (determined using an OCL expression ServerAtMinCapEx-
ists) or removes a vCPU from an application server VM. The undoPreviousAction tactic can be
used in cases where the previous adaptation step must be undone.

5.2.2.3 Strategy

As motivated in the previous section, the purpose of a Strategy is to achieve a high-level Objective.
The objective of a strategy is part of the OverallGoal specified by the system administrator. For exam-
ple, an objective can be described with goal policies or utility function policies. One could also use any
other type of description that can be automatically checked by analyzing the model or monitoring data
from the real system (e.g., based on MAMBA [50]). Note that it is explicitly allowed to have multiple
alternative strategies with the same objective because strategies might differ in their implementation.

The execution of a strategy is triggered by a specific Event that occurs during system operation, e.g.,
when an SLA is violated or a adaptation to increase efficiency is scheduled. Such an event triggers the
execution of the respective strategy with the target to ensure that the objective of the strategy is achieved.
In our approach, events can trigger only one strategy. We assume that events occur sequentially to
avoid concurrency effects, e.g., two strategies operating at the same time but with a conflicting objective.
However, we do not exclude situations where there are conflicting objectives. Such cases can be handled
using strategies with objectives modeled as utility function policies [34]. The respective strategy in such
a situation would try to apply tactics such that the objective of the utility function is achieved.

To achieve its objective, a strategy uses a set of WeightedTactics. A WeightedTactic is a decorator
for a Tactic. Weights are assigned according to the strategy's WeightingFunction. The strategy uses
the weight to determine which tactic to apply next. The weight depends on the current state of the
system, i.e., the weights can change if the state of the system changes. Hence, the strategy might choose
a different tactic in the next adaptation step. Formally, assume

\[ T = \{ t_1, t_2, \ldots, t_m \} \]

is the set of tactics and,

\[ S = \{ s_1, s_2, \ldots, s_n \} \]

is the set of all possible system states.

Then one can think of a mapping

\[ WT \in [T \times S] \rightarrow [0,1] \subseteq \mathbb{R} \]

that assigns a weight to the given tactic \( t \in T \) in the system state \( s \in S \). It is not part of this work and it is intentionally left open to actually specify the mapping in Equation 5.1 in more detail. The idea is to use existing and well-established optimization algorithms or meta-heuristics to determine the weights depending on the current state of the system possibly also considering its previous states stored in a trace. The use of weighted tactics introduces a certain amount of indeterminism at this abstraction level. Having this indeterminism at the strategy level provides for flexibility to find new solutions in case one tactic turns out to be inappropriate for the current system state.

**Example** Figure 5.8 depicts the two strategies of the example algorithm introduced in the beginning of this section, PUSH and PULL, with the objective to improve response times to maintain SLAs, and to ensure efficient resource usage, respectively. The PUSH 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 system’s QoS properties. 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. Determining the weight...
for the PUSH tactic in this scenario is straightforward as it is always 1.0 because no other tactics are defined as part of the strategy.

The PULL strategy is triggered with the objective to optimize resource efficiency, either based on a predefined schedule or directly after the PUSH phase. The PULL strategy has a tactic to reduce the amount of used resources (removeResources). Again, after the execution of the tactic, the 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 resources. In case an SLA violation occurs, the last adaptation must be undone, which is implemented by the undoPreviousAction tactic. Which of these two tactics is chosen is determined based on their weights.
Bibliography


