Modeling Dynamic Virtualized Resource Landscapes*

Nikolaus Huber
Karlsruhe Institute of Technology
Am Fasanengarten 5
Karlsruhe, Germany
nikolaus.huber@kit.edu

Fabian Brosig
Karlsruhe Institute of Technology
Am Fasanengarten 5
Karlsruhe, Germany
fabian.brosig@kit.edu

Samuel Kounev
Karlsruhe Institute of Technology
Am Fasanengarten 5
Karlsruhe, Germany
kounev@kit.edu

ABSTRACT

Modern data centers are subject to an increasing demand for flexibility. Increased flexibility and dynamics, however, also result in a higher system complexity. This complexity carries on to run-time resource management for Quality-of-Service (QoS) enforcement, rendering design-time approaches for QoS assurance inadequate. In this paper, we present a set of novel meta-models that can be used to describe the resource landscape, the architecture and resource layers of dynamic virtualized data center infrastructures, as well as their run-time adaptation and resource management aspects. With these meta-models we introduce new modeling concepts to improve model-based run-time QoS assurance. We evaluate our meta-models by modeling a representative virtualized service infrastructure and using these model instances for run-time resource allocation. The results demonstrate the benefits of the new meta-models and show how they can be used to improve model-based system adaptation and run-time resource management in dynamic virtualized data centers.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques; I.6.5 [**Simulation and Modeling**]: Model Development

Keywords

Meta-model, Resources, Virtualization

1. INTRODUCTION

In today's IT systems, the demand for distributed and more dynamic and flexible data center infrastructures is continuously increasing. This applies particularly to the trend of Cloud Computing which provides a basis for provisioning data center resources on demand in an elastic manner and on a pay-per-use basis. Hence, abstraction layers like virtualization and middleware technologies are implemented to increase the ability of IT systems to react on changes of service workloads and resource demands, i.e., to provision and allocate resources as they are needed, or to consolidate services in order to increase the resource efficiency.

However, introducing new abstraction layers to increase flexibility comes at the cost of increased system complexity. The new abstraction layers provide new reconfiguration possibilities and enlarge the configuration space. However, they also raise new challenges and questions for run-time system adaptation and resource management to ensure Quality-of-Service (QoS). Imagine a simple example with two virtual machines (VMs) of different customers sharing resources and assume the workload of one customer increases, leading to a QoS violation. Questions such as the following arise: Should resources be added or a VM be migrated to solve the problem? Which server is the best target for a VM migration? What impact has virtualization on the applications' QoS?

Instead of costly and possibly imperfect implementation and testing approaches, model-based resource management and capacity planning techniques like [3, 13] can help to answer such questions. However, they are targeted at design-time QoS analyses. In general, such approaches are not applicable at run-time because they abstract the complex infrastructure architecture or do not consider the dynamic aspects important for run-time system adaptation and resource management.

To fill this gap, we present a set of modeling abstractions which are an integral part of the Descartes Meta-Model (DMM) [14], a new meta-model for run-time QoS and resource management in virtualized service infrastructures. These meta-models and their information about the resource landscape and its dynamic aspects form a basis for automated run-time system adaptation and resource management: First, the resource landscape meta-model focuses on describing both physical and virtual resources and the different layers of resources that exist in dynamic data centers. Second, the adaptation points meta-model describes the dynamic parts, i.e., the aspects of the resource landscape which configurable at run-time and their variation range. This avoids modeling each of the various configurations the system might have at run-time. We evaluate these meta-models and concepts by applying them in a case study on automated resource allocation. The results show that the meta-models increase the expressiveness to model resource landscapes and that they improve automated run-time system adaptation and resource management.

*This work was funded by the German Research Foundation (DFG) under grant No. KO 34456-1.

QoSA'12, June 25–28, 2012, Bertinoro, Italy.

The contributions presented in this paper are: i) a resource landscape meta-model capturing properties relevant for run-time performance analysis and resource management of distributed dynamic data centers, ii) an adaptation points meta-model for annotating resource landscape models with information about the dynamic aspects of the system that can be adapted at run-time. We evaluate these contributions in a realistic environment by applying them to a representative virtualized service infrastructure and in experiments on run-time deployment of VMs and dynamic resource allocation.

In Section 2, we discuss related approaches. In Section 3, we present our resource landscape meta-model, the static view of the system. Section 4 introduces the adaptation points model which can be used to describe the dynamic parts of the resource landscape model. Finally, we apply these meta-models in experiments presented in Section 5, before Section 6 concludes this paper.

2. RELATED WORK

The foundations and influences for this work are mainly from two different areas. First, the field of (architecture-level) performance models forms the basis for this work. Second, an important addition to this topic are approaches to describe dynamic systems aspects or dynamic elements of performance (meta-)models.

Over the past decade, a number of architecture-level performance meta-models for describing performance-relevant aspects of software architectures and execution environments have been developed by the performance engineering community [1, 16]. Most prominent examples are the UML SPT profile [17] and its successor the UML MARTE profile [18], both of which are extensions of UML as the de facto standard modeling language for software architectures. Other proposed meta-models include CSM [19], PCM [3], SPE-MM [21], and KLAPER [10]. The common goal of these efforts is to predict the system performance by transforming architecture-level performance models into predictive performance models (e.g. Layered Queuing Networks or (Queuing) Petri Nets) in an automatic or semi-automatic manner. In general, these approaches abstract from the details of the execution environment, modeling hardware as single entities therefore losing important information about the resource landscape and the configuration possibilities.

There exist different approaches on modeling resources and the environment in which they are contained, focussing on different levels of abstraction. A generic approach for modeling the logical and physical resources of a system with UML is [20]. In this work resources are described as entities offering services characterized by Quality-of-Service (QoS) attributes. In addition, this work describes a layered relationship between the client and the resources the client consumes. The concepts presented in this contain interesting ideas but are a too general abstraction for complex resource landscapes. On the other hand, in architecture-level performance models, e.g. the PCM [3], the resources and their environment are described in a coarse-grained fashion. In PCM, the resource landscape consists of containers which provide resources like CPU, HDD or memory. [11] presents an extension of PCM to describe the execution environment of a resource container in more detail. These concepts are very specific, introducing layers like virtualization, JVM and application server using controllers to model the performance relevant parameters of these layers. Because these concepts focus on a single resource container, they are not suitable to describe the whole resource landscape as in modern data centers, e.g., with different VMs contained in a hypervisor.

In general, architecture-level performance models are built during system development and are used at design and deployment time to evaluate alternative system designs and/or predict the system performance for capacity planning purposes. Usually, design alternatives are modeled manually because the models do not support modeling variability. The reason is that these models are generally not designed to be used at system runtime and are not able to completely reflect the variability occurring during runtime of a system. An approach trying to model such dynamic reconfiguration at runtime is SOFA 2.0 [6]. SOFA 2.0 allows component replacement and other reconfigurations but they are restricted to the application level. A more generic approach for the formulation of design alternatives in architecture-level performance models is [15]. The target is to describe the degrees of freedom in a system to automatically find an optimal architecture. Although this approach is limited to design-time optimization, it presents useful concepts and ideas on how to model variability in architecture-level performance models. An approach to describe variants of software product lines is the Variation Point Model [9]. Its concepts to describe variants of variation points is also interesting, but both [15] and [9] describe how variants, i.e., different instances of the same core concept might look like, whereas the meta-models presented in this work focus on the configuration range of a single model instance.

With the use of formal ways to specify system and software architectures it is possible to use this level of abstraction to describe the dynamic aspects of the architecture and also how the system architecture and/or its configuration can adapt to changes in the environment. One can find many different approaches in this area and a survey of such formal specification approaches is given by Agnew et al. [4]. Although these approaches are suitable to describe adaptation at the architecture level, they usually do not provide means to describe how to react on changes. More specifically, it is difficult to describe, how adaptation should be executed. As influential examples for approaches also describing the actual system adaptation process are Service Activity Schemas (SAS) [8] and software architecture-based adaptation [7].

In summary, each of the presented existing approaches on architecture-level performance models, resource landscapes and dynamic system reconfiguration and adaptation contains valuable aspects to consider in our approach. However, no single exhausting approach covering all aspects exists.

3. RESOURCE LANDSCAPE (STATIC VIEW)

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 paper we focus on the computational infrastructure. Modeling other aspects like storage or network infrastruc-

ture with switches or storage services are briefly considered in this paper, but they are part of future work. In Section 4, we introduce the meta-model to annotate this static view of the resource landscape with the dynamic aspects, i.e., the adaptable parts of the system.

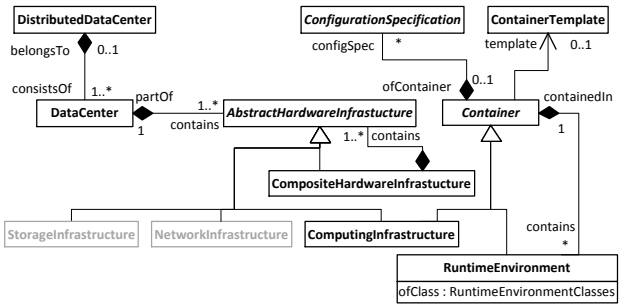

Figure 1: Resource landscape meta-model.

Figure 1 depicts an high-level view and the overall structure of the resource landscape meta-model as an UML class diagram. The root entity comprising 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 `StorageInfrastrucutre`. 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 3.1. Meta-modeling `StorageInfrastructure` and `NetworkInfrastructure` in more detail is out of this paper's scope and part of future work.

3.1 Containers and Containment Relationships

A reoccurring pattern in distributed dynamic data centers is the nested containment of system entities. For example, imagine data centers containing servers, which might contain a virtualization platform and VMs, again containing an operating system, containing a middleware layer and so on. This leads to a tree of nested entities which can change during runtime (e.g., when a VM is migrated). Because of this flexibility one single execution environment consisting of similar, reoccurring elements can have multiple different configuration states at run-time. However, current architecture-level performance models do not cover these hierarchy of resources and its configurations which we want to model.

The central entity of the resource landscape meta-model depicted in Figure 1 is the abstract entity `Container`. The `Container` has a property `configSpec` to specify its configuration (which will be explained in Section 3.3) and a property `template` referring to a `ContainerTemplate` (which will be explained in Section 3.4). Most important is that a `Container` has an explicit reference to the entity `RuntimeEnvironment` to model that it can contain further containers. i.e., to model the tree-like structure of entities explained above. 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 3.2.

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

3.2 Classes of Runtime Environments

We distinguish six general classes of runtime environments which are listed in Figure 2. 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:

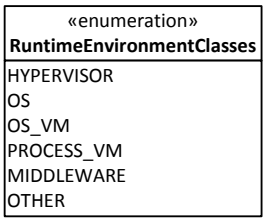

Figure 2: Different runtime environment classes.

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).

3.3 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 3(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 `bandwith` 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.

3.4 Container Types

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.

Our solution is to use a `ContainerRepository` (see Figure 3(b)). One can specify `ContainerTemplates` and collect them in the `ContainerRepository`. 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 1). 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. This way, only deltas to the container template must be modeled and not all configurations for all containers. Container templates are also beneficial later when describing the dynamic parts of the system. With container templates, the configurable dynamic parts must be specified only one, namely for the container template and not for each container. 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., an additional meta-model for container types to create instances of container types. However, this would require that a provider of a container (e.g. a virtualization platform vendor) must be familiar with meta-modeling.

3.5 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. 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 model [2]. 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 an the service assembled to a system (here named `AssemblyContext`) with a container instance of the resource landscape model. The meta-model for modeling services is also developed as part of the DMM. However, the details would go beyond the scope of this work and are given in [14]. 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

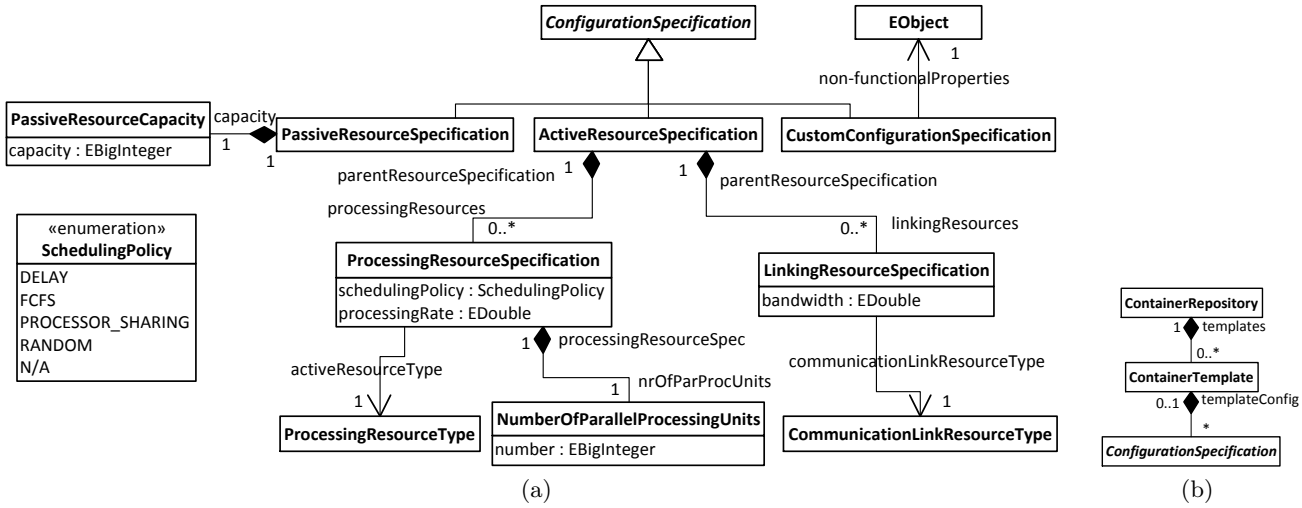

Figure 3: Types of resource configurations (a) and the container repository (b)

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`. Because the description of services and the service architecture is not in the scope of this paper, we refer to [14] for more technical details.

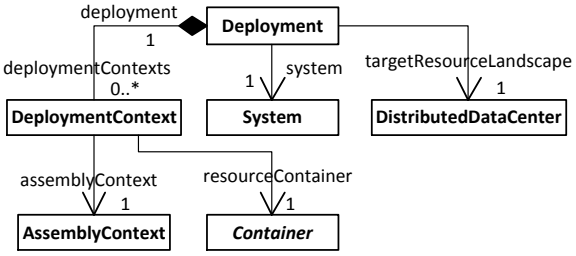

Figure 4: The deployment model.

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 [11]. In this approach the resource demands can be mapped to interfaces provided by the resources. Controllers in the layers providing these interfaces take care of mapping the resource demands, e.g., adding overheads occurring in this layer.

4. ADAPTATION POINTS (DYNAMIC VIEW)

So far, the model was focused on the static aspects of the system. However, 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. Therefore, we introduce an additional meta-model which addresses this flexibility and variability.

The *Adaptation Points* meta-model is an addition to the static sub-models of DMM. In the following, we use the adaptation points meta-model to describe which parts of the resource landscape model are variable and can be adapted 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 using the adaptation points model. Figure 5 depicts the overview of the adaptation points meta-model.

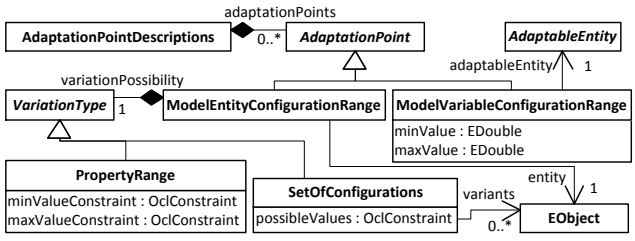

Figure 5: Adaptation points meta-model.

The specifications, i.e., the annotations of the variable elements of a resource landscape model instance are collected 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. Note that in the DMM model variables are meta-model classes with an attribute that is variable, i.e., it may change during run-time (e.g., workload) or it can be changed (e.g., number of virtual CPUs, `NumberOfParallelProcessingUnits`, see Fig. 3).

These two types of `AdaptationPoints` are modeled as `ModelVariableConfigurationRange` (a) and `ModelEntityConfigurationRange` (b), respectively, specifying the range in which the resource landscape 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. All meta-model entities which are adaptable at run-time inherit from this type. Hence, at the meta-model level, all entities of type `AdaptableEntity` are considered as adaptable elements. However, on the model instance level, they are only considered to be adaptable if they are actually annotated by a `AdaptationPoint`. The reason is that even if a system has adaptation points there might be an instance of this system 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 resource landscape 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 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 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 and are collected in the `SetOfConfigurations`. For example, this set collects the references to the different runtime environment instances a VM can be deployed on.

In summary, this meta-model concentrates on the description of all possible configurations one single instance of a dynamic system might have. It is not intended to describe all possible instance variants a dynamic system might take (see Section 2 for more details).

5. EVALUATION

In this section we present example instances of the previously introduced meta-models which describe the resource landscape and adaptation points of a realistic data center. In experiments, we analyze the models using simulation and leverage their encapsulated information to improve run-time system adaptation and resource management to demonstrate the advantage of the resource landscape model compared to other architecture-level performance models.

First, we will present the resource landscape and its model instance of a realistic data center (Section 5.2) and explain the corresponding adaptation points (Section 5.3) before we show experimental results and discuss our approach.

5.1 Experiment Environment

The data center we will now model consists of six compute nodes from our local cluster environment (see Fig. 6). Each compute node is equipped with two Intel Xeon E5430 4-core CPUs running at 2.66 GHz and 32 GB of main memory. The machines are connected by a 1 GBit LAN. On five of these compute nodes we run XenServer 5.5 as the virtualization layer. The VMs are initially equipped with eight virtual CPUs (a VCPU corresponds to a core). The sixth compute node is not virtualized because it hosts the database.

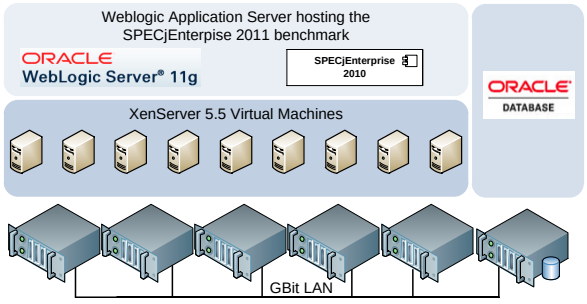

Figure 6: Experiment environment consisting of six virtualized cluster nodes and a native database server.

On top of this infrastructure and inside VMs we execute the SPECjEnterprise2010 benchmark[1], a representative, state-of-the art application we have successfully modeled and used in the context of automated model extraction [5] and dynamic resource allocation [12]. We think of the VMs as SPECjEnterprise2010 instances that belong to different customers and each customer has its own performance requirements, stipulated as Service Level Agreement (SLA). To maintain SLAs, the platform must scale and provide enough resources in situations where, e.g., the workload varies. Simultaneously, the resources of the data center should be used as efficient as possible.

5.2 Modeling Dynamic Data Centers with the Resource Landscape Model

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

Figure 7 shows the resource landscape model instance of our data center and the configuration of the resource containers. It reflects the hierarchy of the resource containers as well as their performance-relevant configuration. Note that the depicted model instance in Figure 7 is incomplete, showing only three of the six compute nodes due to space constraints. The other compute nodes not shown are modeled the same way. Note that *Cn*6 contains no hypervisor runtime environment, i.e., it is a native system.

The root element is the `DistributedDataCenter` *DDC* which contains the *LocalDC*. Within the *LocalDC* we model our local cluster environment called *AcamarCluster* as `CompositeHardwareInfrastructure` to group all contained compute nodes. Such information is useful for, e.g., VM migration because migrating a VM might be limited for technical reasonsn, e.g., the NFS share is only accessible for all compute nodes within the cluster. Each compute node refers to the *CnTemplate* which specifies the resource configuration of the respective compute node. Similarly, the nested *Xen* and *VM* `RuntimeEnvironments` refer to their type-specific templates.

This example shows the advantages of our meta-model. With the template mechanism it is possible to have multiple instances of the same type in the model instance. If one changes the configuration of the template, all instances in the model referring to the changed template are affected by the change. However, with the override mechanism as described in Section 3.4, it is also possible to have individual specifications for each instance of, e.g., a VM. To undo the individual specification of a container, one can simply delete this specific property to fall back to the template configuration. Furthermore, one can model performance-influences on each layer of the resource stack (the virtualization layer in this example). Further layers like the operating system have been omitted to keep the example more clear. The performance properties of the application layer, i.e., the services deployed on these containers are be modeled with a separate meta-model but are not depicted here.

5.3 Modeling Dynamics with the Adaptation Point Model

We now show how to use the adaptation points meta-model to specify the variable dynamic parts of the resource landscape. To this end, we create a adaptation points model instance that annotates the resource landscape model instance. The variable resources and entities we consider here are i) the number of virtual CPUs of a VM and ii) the number of application server instances, i.e., VMs.

Corresponding to these variable elements, the adaptation points model instance contains two different adaptation points, one `ModelVariableConfigurationRange` and one `ModelEntityConfigurationRange`. *VariableVcpu* describes that the number of virtual CPUs of the VM is variable between two and eight. Furthermore, *VariableVcpu* refers to the `AdaptableEntity` instance *Cores*, the actual model parameter to vary. It is important to note that the referenced model entity is actually the entity contained in the configuration specification of the VM container template. This way the model instance describes that all VCPUs of all VMs referring to this template can be varied. This is a major advantage of the template mechanism of the resource landscape model.

The other adaptation point of the resource landscape is *VariableAppServerInstances*. It refers to the computing infrastructure template *ComputeNode* and has two attributes that specify OCL constraints:

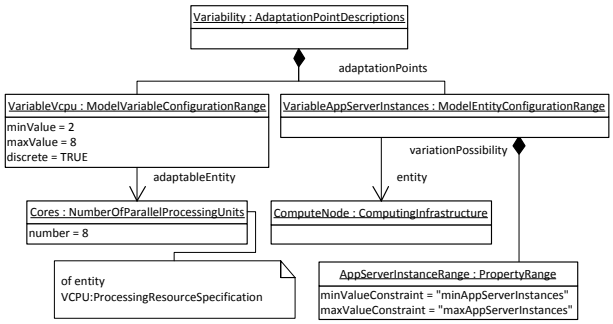

Figure 8: Adaptation points meta-model instance annotating the resource landscape model.

```
context ModelEntityConfigurationRange
inv minAppServerInstances:
Container.allInstances()
        ->select(c | c.template = self.entity)
                ->size()>1 and
Container.allInstances()
        -> select (c | c.template = self.entity)
        -> forAll(c | c.contains.contains
                ->size() = 1);

context ModelEntityConfigurationRange
inv maxAppServerInstances:
Container.allInstances()
        ->select(c | c.template = self.entity)
                ->size()<=3 and
Container.allInstances()
        -> select (c | c.template = self.entity)
        -> forAll(c | c.contains.contains
                ->size() = 1);
```

The OCL constraints ensure that there is at least one application server (i.e. one VM running an application server) and three at most (as an example constraint). The OCL constraints query for all `Container` instances and select those which refer to the same template as the `ModelEntityConfigurationRange`. The resulting number must be above the minimum and below the maximum. Note that for the correct evaluation of the OCL constraints the scope of the OCL engine must be set to the `ModelEntityConfigurationRange` instance which actually refers to the model instance entity to be evaluated. This is important because with the scope the modeled adaptation variability can be restricted.

5.4 Experiment Results

To evaluate our resource landscape model and demonstrate its advantages we conduct an experiment showing how the model and its analysis results can improve system adaptation and resource management. To this end, we use the models in two scenarios concerning run-time (re)deployment of virtual machines and dynamic resource (re)allocation. To analyze the model, we transformed our model instances to the Palladio Component Model [3] and then simulate these models with the Palladio Simulator[2] to obtain resource usage and service response time predictions. The Palladio Simulator provides simulation results for detailed analyses or analytical solving techniques based on layered queuing networks in case time of time constraints at run-time. The

[2] `http://www.palladio-simulator.org`

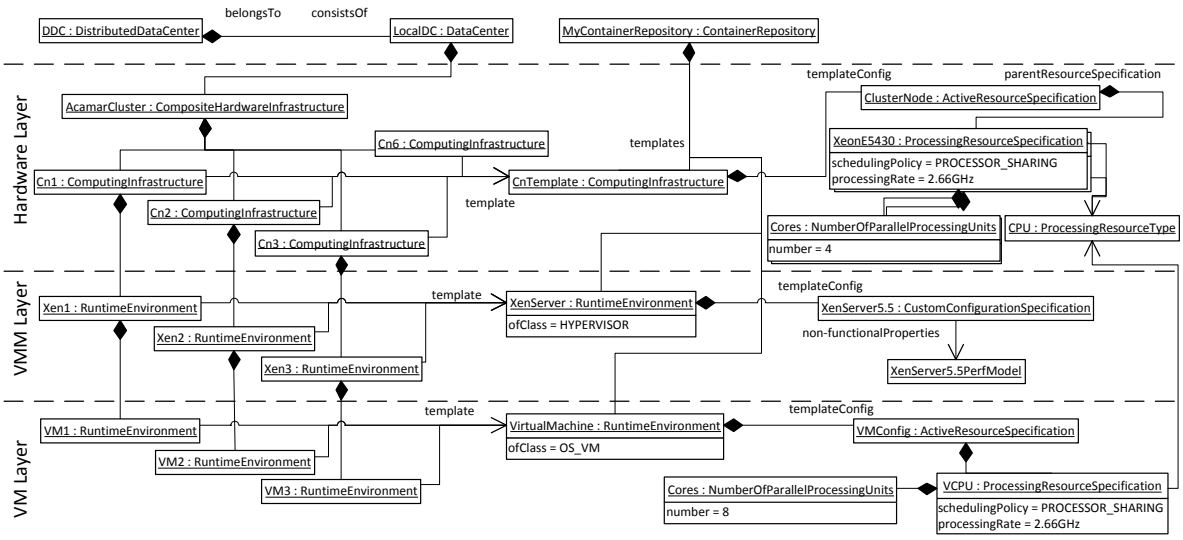

Figure 7: Resource landscape meta-model instance of data center used in the experiments.

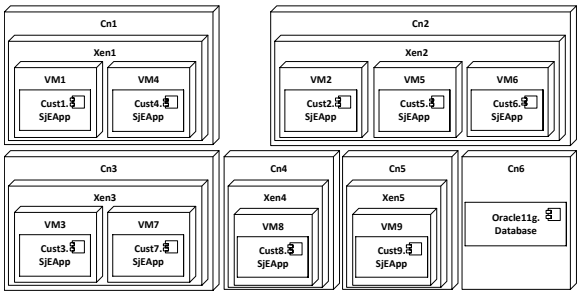

Figure 9: Initial deployment of the SPECjEnterprise2010 benchmark customer instances (`Depl_0`).

model's structure and parameters are based on the model we obtained with measurements in [12].

5.4.1 (Re)deployment of Virtual Machines

We start with a scenario consisting of nine independent instances of the SPECjEnterprise2010 benchmark, each instance for a separate customer. The initial deployment (reffered to as `Depl_0`) of these benchmark instances in the data center is depicted in Figure 9. Besides this resource landscape model, we also modeled the services of the SPECjEnterprise2010 benchmark in more detail as well as the use profile of the customers. We omitted a detailed description of these meta-models because they are not in the focus of this work and more details are given in [14]. Assume that `Cust2, Cust6` and `Cust9` are gold customers and an SLA guarantees a response time below 20ms for the default workload whereas `Cust1, Cust3, Cust4, Cust5` and `Cust8` are a silver customers with guaranteed response times below 40ms. Another constraint is that all compute node utilizations must be below 90% to avoid heavy response time fluctuations at high system load. In this initial deployment `Depl_0` depicted in Figure 9 and with the default workload (`Default`), the requirements are fulfilled, i.e., the system is in a valid state (see column `Depl_0 (Default)` in Table 1).

Now assume that the workload of `Cust1` doubles. The simulation results shown in column `Depl_0 (High)` of Table 1 for the increased workload show that the utilization of the compute node $Cn1$ would be above the limit of 90%, requiring a new deployment of the VMs such that requirements are maintained. Intuitively, one would at first try to migrate the VM with the higher load to the part of the system with the least utilized resources (e.g., $Cn6$ or $Cn2$). However, we will see that querying the model and using its architectural information leads to different results.

The model and its analysis results show that $Cn6$ has plenty of resources. However, migrating any of the VMs to $Cn6$ is not an option because $Cn6$ is not virtualized, i.e., it has no runtime environment `HYPERVISOR` which could contain an `OS_VM`. Migrating $VM4$ to $Cn5$ is impossible because this would again lead to a violation of the utilization threshold ($67.5\% + 25.7\% > 90\%$) as would migrating $VM1$ to any other compute node. Hence, three options remain, migrating $VM4$ to either $Cn2$ (`Depl_1`), $Cn3$ (`Depl_2`) or $Cn4$ (`Depl_3`). We simulated these scenarios and the predicted utilization for these cases are below the threshold, i.e., regarding the resource usage we have three possible adaptation options. But which one is better? Further constraints for the adaptation options are the SLAs stipulated in the contracts with the customers. Since the workload of these two customers did not change, system adaptation should have no implications on their SLAs. However, the predicted response times (see Table 2) show that migrating $VM4$ has a significant impact on the response time of the customer whose VM has to share its resources. The results show that adaptation option `Depl_1` violates the SLA of the gold customer whereas SLAs are fine for `Depl_2`. `Depl_3` has no effect on the response times of the gold customers but the SLA of the silver customer `Cust8` is violated. Therefore, the only option to reconfigure the system without using additional resources is to migrate $VM4$ to $Cn3$ (`Depl_2`).

5.4.2 Dynamic Resource (Re)allocation

In this scenario we apply the adaptation points model in a resource allocation algorithm for virtualized environments. We used this resource allocation algorithm [12] in an eval-

Table 1: Simulated utilization results for different workload situations and adaptation options.

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

Table 2: Simulated response times for the service `purchase` for the two deployment options.

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

uation consisting of four scenarios with different workloads to trigger and evaluate the resource allocation algorithm. Previously, this algorithm was purely written in Java and directly manipulating the model instance, i.e., the knowledge which parts of the model can be varied and how to vary them was captured within Java code. Now we can use the adaptation points model and interpret it to change the resource landscape model.

The results depicted in Figure 10 illustrate how the resource landscape model instance of the data center changes in the four scenarios as a reaction on the changing workloads. The adaptation effect is reflected by the number of instances of AppServer and VCPU, the two adaptation points of the resource landscape model. As one can see, the AppServers and VCPUs increase as the workload increases, i.e., the model instance is adapted to the change in the environment. For example, with six times the standard workload, three AppServer instances with eleven VCPUs in total are necessary to handle the load.

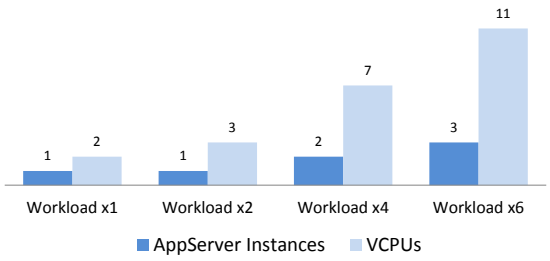

Figure 10: Adaptation impact on the model instance in the four different scenarios from [12].

5.5 Summary and Discussion

Our experiments show that the meta-models are suitable to describe distributed dynamic data centers and that the information captured by the resource landscape model is important for run-time system adaptation to guide the adaptation process for several reasons.

The resource landscape model reflects the various layers of resources and the landscape of modern data centers that impact performance analysis and resource management. Other architecture-level performance models do usually not capture information about the layers of the resources. Generally, they assume a flat hierarchy of resource containers because the hierarchy is not relevant at design-time or for offline use. However, at run-time and especially for autonomic and self-aware system adaptation and resource management, such information is crucial. The results show that modeling the resource landscape with its hierarchy contains valuable information to, e.g., exclude migration targets or find the most suitable target. Furthermore, with the hierarchy of nested resource containers we can reflect the effect of resource sharing (the main benefit of virtualization and a core aspect in dynamic data centers) and detect performance bottlenecks or resource inefficiencies on the hardware level instead of just inside VMs. Finally, with the different layers of resources we can integrate the performance-relevant properties of the different layers in the QoS analyses.

The example meta-model instance of the adaptation points shows that it is necessary to know the details of the annotated resource landscape model instance to specify the adaptation points. However, the reason why we did not consider adaptation on the resource landscape meta-model level is that we want to describe the adaptation variability for specific meta-model instances because the model instances exist at run-time and must be changed online. This is a difference to the approach in [15] which focuses on the meta-model level to describe which variants of model instances can be created at design time. By using the adaptation points model for system adaptation, the resource allocation algorithm's actions can refer to the dynamic elements of the model instead of working on the model directly. This separates the knowledge about *what* can be reconfigured from *how* to execute the adaptation. For example, one must not know where and how to change a resource landscape model instance. This information is encapsulated in the adaptation points model. Another benefit of this separation is that the information about adaptation points can be re-used in different resource allocation algorithms.

A further strength of the meta-models presented in this paper is that the static and dynamic elements can be managed independently. By providing the template repository, it is also possible to specify adaptation points descriptions for templates. Considering this when creating the adaptation point descriptions, it is possible to reuse these descriptions together with the templates in different meta-model instances. This eases also model evolution and maintainability.

In summary, the advantage of the presented models compared to other design-time architecture-level performance models is the focus on the dynamic and resource layers of distributed dynamic data centers. These aspects are captured

in two separate meta-models to separate them for maintainability and usability reasons. The example demonstrates how this information can be successfully used to improve performance and resource management at run-time.

6. CONCLUSIONS

With technologies like virtualization and Cloud Computing, modern data centers are becoming increasingly dynamic and complex. Novel resource landscape meta-models reflecting this variability and complexity are required to enable model-based run-time system adaptation and resource management.

In this paper, we presented a meta-model to describe the resource landscape and its different layers of abstractions in distributed dynamic data centers (static view). Second, we introduced a meta-model for annotating resource landscape model instances to describe the aspects adaptable at run-time (dynamic view). Finally, we applied our concepts to experiments on run-time deployment of VMs and dynamic resource allocation.

The high-level rationale of these comparably detailed meta-models and the reason why we chose this level of granularity was that we experienced limitations of current architecture-level performance models when applied at run-time (see Section 2). For example, at run-time it is necessary to model the dynamic aspects and the hierarchy of the resource landscape and its various layers which is difficult with design-time models.

The results of our evaluation demonstrate that our meta-models are suitable to describe the details of the static and dynamic aspects with increased expressiveness. We showed that it is important to capture all details at the suggested level of granularity to achieve improved analysis accuracy and thereby obtain better decisions in model-based run-time system adaptation and resource management techniques.

As part of our ongoing work, we are integrating the proposed meta-model with meta-models for managing resource landscapes of distributed virtualized data centers. We plan to evaluate the applicability of the Descartes Meta-Model in an extensive case study. Further targets are also to integrate storage and network infrastructure models as well as other non-functional properties.

7. REFERENCES

[1] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni. Model-Based Performance Prediction in Software Development: A Survey. *IEEE TSE*, 2004.

[2] S. Becker, J. Happe, and H. Koziolek. Putting Components into Context: Supporting QoS-Predictions with an explicit Context Model. In *WCOP'06*, 2006.

[3] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Sys. and Softw.*, 82:3–22, 2009.

[4] J. Bradbury, J. Cordy, J. Dingel, and M. Wermelinger. A survey of self-management in dynamic software architecture specifications. In *ACM SIGSOFT Workshop on Self-managed systems*, 2004.

[5] F. Brosig, N. Huber, and S. Kounev. Automated Extraction of Architecture-Level Performance Models of Distributed Component-Based Systems. In *26th IEEE/ACM International Conference On Automated Software Engineering (ASE 2011)*, 2011.

[6] T. Bures, P. Hnetynka, and F. Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *Software Engineering Research, Management and Applications*, 2006.

[7] S. Cheng, D. Garlan, B. Schmerl, J. Sousa, B. Spitznagel, P. Steenkiste, and N. Hu. Software architecture-based adaptation for pervasive systems. *Trends in Network and Pervasive Comp. (ARCS'02)*.

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

[9] H. Gomaa and D. Webber. Modeling adaptive and evolvable software product lines using the variation point model. In *Int. Conf. on System Sciences*, 2004.

[10] V. Grassi, R. Mirandola, and A. Sabetta. Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach. *J. of Sys. and Softw.*, 80(4):528–558, 2007.

[11] M. Hauck, M. Kuperberg, K. Krogmann, and R. Reussner. Modelling Layered Component Execution Environments for Performance Prediction. In *Proc. of CBSE 2009*.

[12] N. Huber, F. Brosig, and S. Kounev. Model-based Self-Adaptive Resource Allocation in Virtualized Environments. In *6th Int. Symp. on Softw. Eng. for Adaptive and Self-Managing Systems (SEAMS)*, 2011.

[13] S. Kounev. Performance Modeling and Evaluation of Distributed Component-Based Systems using Queueing Petri Nets. *IEEE Transactions on Software Engineering*, 32(7):486–502, 2006.

[14] S. Kounev, F. Brosig, and N. Huber. Descartes Meta-Model (DMM). Technical report, Karlsruhe Institute of Technology (KIT), 2012. To be published.

[15] A. Koziolek and R. Reussner. Towards a generic quality optimisation framework for component-based system models. In *Proc. of CBSE 2011*.

[16] H. Koziolek. Performance Evaluation of Component-based Software Systems: A Survey. *Performance Evaluation*, 67(8):634–658, August 2010.

[17] OMG. UML Profile for Schedulability, Performance, and Time (SPT), v1.1, Jan. 05.

[18] OMG. UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE), v1.0. `http://www.omg.org/spec/MARTE/1.0/PDF`, Nov. 20.

[19] D. Petriu and M. Woodside. An intermediate metamodel with scenarios and resources for generating performance models from UML designs. *Softw. and Syst. Modeling*, 6(2):163–184, 2007.

[20] B. Selic. A generic framework for modeling resources with UML. *Computer*, 33(6):64–69, 2000.

[21] C. U. Smith, C. M. Lladó, V. Cortellessa, A. Di Marco, and L. G. Williams. From UML models to software performance results: an SPE process based on XML interchange formats. In *Proceedings of WOSP 2005*, pages 87–98. ACM Press, 2005.