Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

# Architecture-level software performance abstractions for online performance prediction[☆]

Fabian Brosig[*], Nikolaus Huber, Samuel Kounev

*Karlsruhe Institute of Technology, Am Fasanengarten 5, Karlsruhe, Germany*

## H I G H L I G H T S

- Discussion between online and offline performance prediction scenarios.
- Modeling performance-relevant service behavior at different levels of detail.
- Modeling parameter dependencies specifically for use at run-time.
- Evaluation based on the SPECjEnterprise2010 standard benchmark.

A R T I C L E   I N F O

*Article history:*
Received 15 October 2012
Received in revised form 5 June 2013
Accepted 12 June 2013
Available online xxxx

*Keywords:*
Online performance prediction
Architecture-level performance meta-model
Parameter and context dependencies

A B S T R A C T

Modern service-oriented enterprise systems have increasingly complex and dynamic loosely-coupled architectures that often exhibit poor performance and resource efficiency and have high operating costs. This is due to the inability to predict at *run-time* the effect of workload changes on performance-relevant application-level dependencies and adapt the system configuration accordingly. Architecture-level performance models provide a powerful tool for performance prediction, however, current approaches to modeling the context of software components are not suitable for use at run-time. In this paper, we analyze typical online performance prediction scenarios and propose a performance meta-model for (i) expressing and resolving parameter and context dependencies, (ii) modeling service abstractions at different levels of granularity and (iii) modeling the deployment of software components in complex resource landscapes. The presented meta-model is a subset of the Descartes Meta-Model (DMM) for online performance prediction, specifically designed for use in *online* scenarios. We motivate and validate our approach in the context of realistic and representative online performance prediction scenarios based on the SPECjEnterprise2010 standard benchmark.

## 1. Introduction

Modern enterprise software systems are increasingly complex and dynamic. They are typically composed of loosely-coupled services that operate and evolve independently. The increased flexibility gained through the adoption of, e.g., paradigms like service-oriented architecture, comes at the cost of higher system complexity. Managing system resources in such environments to ensure acceptable end-to-end Quality-of-Service (QoS, e.g., performance and availability) while at the same time optimizing resource utilization is a challenge. This is due to the inability to keep track of workload changes and application-level dependencies and their effect on the system QoS. Service providers are often faced with questions such as: What performance would a new service deployed on the infrastructure exhibit and how much resources should be

[☆] This work was funded by the German Research Foundation (DFG) under grant No. KO 34456-1.
[*] Corresponding author. Tel.: +49 721 608 45766.
*E-mail address:* fabian.brosig@kit.edu (F. Brosig).

0167-6423/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.scico.2013.06.004

allocated to it? How should the system configuration be adapted to avoid performance issues or inefficient resource usage arising from changing customer workload parameters? Answering such questions requires the ability to predict at *run-time* how the performance of running services would be affected if the workload or the system configuration changes. We refer to this as *online performance prediction* [1].

Existing approaches to online performance prediction (e.g., [2–5]) are based on stochastic performance models such as (layered) queueing networks or queueing Petri nets. Such models, often referred to as *predictive* performance models, normally abstract the system at a high level without explicitly taking into account its software architecture (e.g., flow of control and dependencies between software components) and configuration. 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 and configuration exist in the literature, however, such models are intended for use at design-time (e.g., [6–9]). 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 used at design time to evaluate alternative system designs and/or predict the system performance for capacity planning purposes.

While architecture-level performance models provide a powerful tool for performance prediction, they are typically expensive to build and 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* [10] 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.

We argue that there are fundamental differences between offline and online scenarios for performance prediction. This leads to different requirements on the underlying performance abstractions of the system architecture and the respective performance prediction techniques suitable for use at design-time vs. run-time. In particular, the type and amount of data available as a basis for model parameterization and calibration at system design-time vs. run-time is different. Furthermore, current approaches to modeling the component context in architecture-level performance models are not suitable for use at run-time since they do not provide enough flexibility in the way parameter dependencies can be expressed and resolved.

In this paper, we analyze the above-mentioned issues when trying to use classical architecture-level modeling approaches in an online scenario and propose a new meta-model for online performance prediction. This involves (i) a novel approach to model the component context and parameter dependencies specifically for use at run-time, (ii) a new approach to model performance-relevant service behavior at different levels of detail, and (iii) a meta-model to describe the resource environment and the deployment of the software system. The presented modeling abstractions are subsets of the Descartes Meta-Model (DMM) [11], a new meta-model for run-time QoS and resource management in virtualized service infrastructures. In this paper, we focus on performance modeling abstractions for the application architecture-level. For details on modeling virtualized infrastructures and system adaptation with DMM, we refer to [12,13]. We motivate and validate our meta-model elements in the context of the industry-standard SPECjEnterprise2010 benchmark,[1] which provides a set of realistic and representative application scenarios such as customer relationship management, manufacturing and supply chain management.

In summary, the contributions of this paper are: (i) analysis of typical online performance prediction scenarios and the requirements on online modeling approaches, (ii) novel modeling abstractions and concepts for expressing and resolving parameter and context dependencies providing increased flexibility at run-time, (iii) a new approach to model performance-relevant service behavior at different levels of detail, (iv) detailed evaluation of the suitability of the proposed abstractions in the context of a set of representative real-life scenarios. To the best of our knowledge, no similar performance modeling approach at the architecture-level providing the level of flexibility achieved through the proposed abstractions exists in the literature.

This paper extends our previous work [14] in the following points. We provide an analysis presenting the different requirements for offline respectively online scenarios on the underlying abstractions for performance prediction. The analysis captures several differences including meta-model structure, type and amount of available monitoring data, trade-off decisions and different degrees-of-freedom. Furthermore, we added a brief description of how we model the resource landscape with DMM [11]. Additionally, we provide new evaluation scenarios (i) showing the differences between the different ways to model service behavior abstractions and (ii) illustrating the performance-relevance of parameter dependencies and how they can be represented in a performance model.

The remainder of this paper is organized as follows. Section 2 describes the different requirements for offline respectively online scenarios on the performance abstractions. Section 3 discusses current approaches to modeling parameter and context dependencies in performance models as implemented in the Palladio Component Model (PCM). In Section 4, we present the novel modeling abstractions and validate their suitability in the context of representative real-life scenarios. We summarize the results from the evaluation of the feasibility and accuracy of the proposed approach in Section 5. Finally, we review related work in Section 6 and conclude in Section 7.

[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.

## 2. Design-time vs. run-time models

We argue that there are some fundamental differences between offline and online scenarios for performance prediction. This leads to different requirements on the underlying performance abstractions of the system architecture and the respective performance prediction techniques 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.

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

In contrast, at run-time, the system design (i.e., architecture) is relatively stable and the main goal of online performance prediction is to predict the impact of dynamic changes in the environment (e.g., changing workloads, 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 do not 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

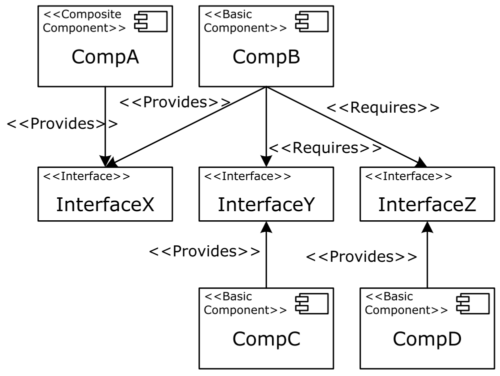


**Fig. 1.** Components providing and requiring interfaces.

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.

## 3. Example of an architecture-level performance modeling language

One of the most advanced architecture-level performance modeling languages, in terms of parameterization and tool support, is the Palladio Component Model (PCM) [6]. In this paper, we use PCM as an example of a mature component-based performance meta-model to motivate and discuss the issues when trying to use classical architecture-level modeling approaches in an online scenario. To make the paper self-contained, we first provide a brief overview of PCM.

PCM provides a meta-model designed to support the prediction of extra-functional properties of component-based software architectures. It is focused on design-time performance analysis, i.e., enabling performance predictions early in the development lifecycle to evaluate different architectural design alternatives. The performance behavior of a component-based software system is a result of the assembled components' performance behavior. In order to capture the behavior and resource consumption of a component, four factors are 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 the way the component is used, i.e., its usage profile, and the execution environment in which the component is running are taken into consideration.

PCM models are divided into five sub-models: The *repository model* consists of interface and component specifications. A component specification defines which interfaces the component provides and requires. For each provided service, the component specification contains an abstract description of the service's internal behavior. The *system model* describes how component instances from the repository are assembled to build an entire system. The *resource environment model* specifies the execution environment in which the system is deployed. The *allocation model* describes the mapping of component instances from the system model to resources defined in the resource environment. The *usage model* describes the user behavior. It captures the services that are called, the frequency (workload intensity) and order in which they are invoked, and the input parameters passed to them.

*Component model and system model.* A component may be either a *basic* (i.e., atomic) component or a *composite* component. Fig. 1 shows an example with 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` respectively `CompD`. A composite component may contain several child component instances assembled through so-called *assembly connectors* connecting required interfaces with provided interfaces. A component-based *system* is modeled as a designated composite component that provides at least one interface. An example of how a composite component is assembled is shown in Fig. 2. Component `CompA` comprises three instances of basic components introduced in Fig. 1 connected according to their provided and required interfaces.

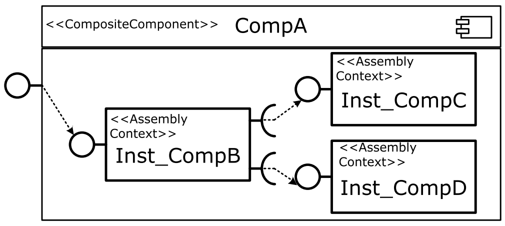

<<CompositeComponent>> CompA
<<Assembly Context>> Inst_CompB
<<Assembly Context>> Inst_CompC
<<Assembly Context>> Inst_CompD

**Fig. 2.** Assembly of the composite component.

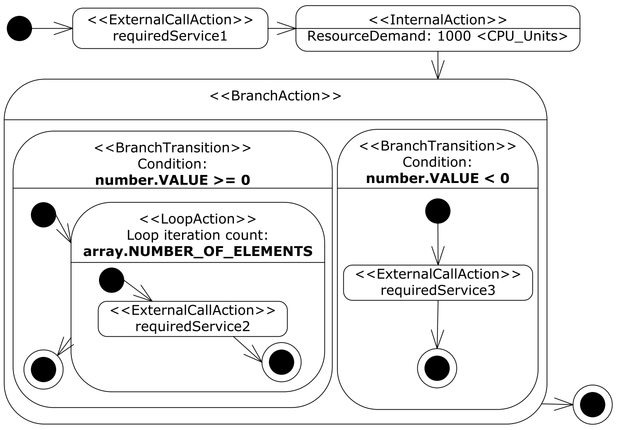

<<ExternalCallAction>> requiredService1
<<InternalAction>> ResourceDemand: 1000 <CPU_Units>
<<BranchAction>>
<<BranchTransition>> Condition: number.VALUE >= 0
<<BranchTransition>> Condition: number.VALUE < 0
<<LoopAction>> Loop iteration count: array.NUMBER_OF_ELEMENTS
<<ExternalCallAction>> requiredService2
<<ExternalCallAction>> requiredService3

**Fig. 3.** RDSEFF of service with signature `execute(int number, List array)` (cf. [6]).

*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) [6]. 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. An example of a RDSEFF for the service `execute(int number, List array)`[6] is shown in Fig. 3. Starting with an `ExternalCallAction` to a required service and an `InternalAction`, 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, the performance behavior abstraction is encapsulated in the component type specification, 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 of modern enterprise software systems 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.

## 4. Performance meta-model

In this section, we present a new performance meta-model specifically designed for use in online scenarios. We motivate and validate our approach in the context of a realistic and representative online performance prediction case study based on the industry-standard SPECjEnterprise2010 benchmark.

Starting with an overview on SPECjEnterprise2010, we present (i) our meta-model's flexible service behavior abstractions, (ii) a novel approach to model parameter dependencies and (iii) an introduction to our infrastructure resource meta-model [12,11]. For each of the three model abstractions, we first provide a motivation using SPECjEnterprise2010 scenarios, then describe the modeling approach at a high-level, followed by a description of the semantics of the modeling entities and

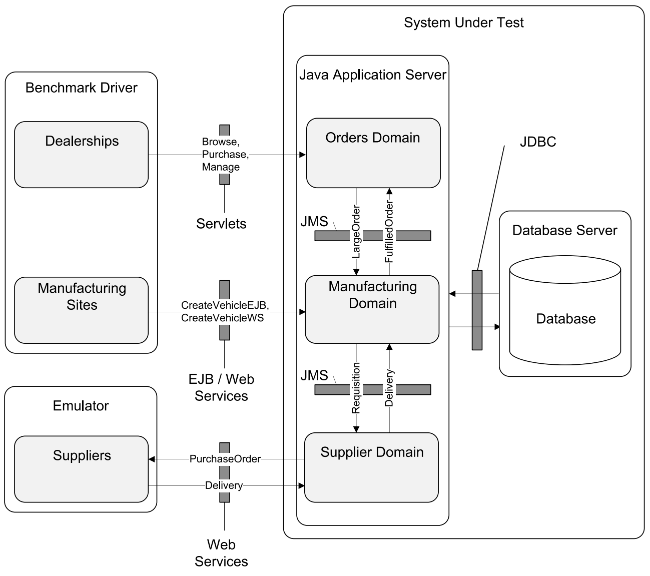


**Fig. 4.** SPECjEnterprise2010 architecture [15].

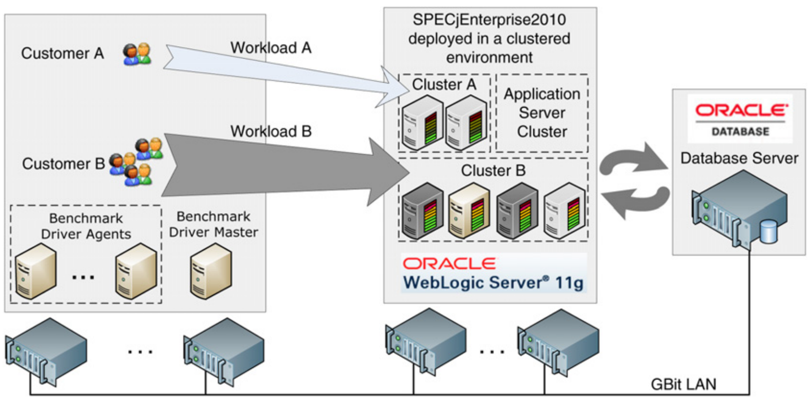


**Fig. 5.** SPECjEnterprise2010 deployment.

an evaluation that is focused on the introduced abstractions. Note that we evaluate the entire modeling approach later in Section 5.

### 4.1. *Online performance prediction scenario*

SPECjEnterprise2010 is a Java EE benchmark for measuring the performance and scalability of Java EE-based application servers. It implements a business information system of 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.

Fig. 4 depicts the architecture of the benchmark. We refer the reader to [15] for further details. We consider a scenario where a set of customers are running their applications in a distributed execution environment. In our case, each customer is running an instance of SPECjEnterprise2010 tailored to a particular type of workload (e.g., CRM, SCM). Each benchmark application instance is deployed in one application server cluster in our deployment environment depicted in Fig. 5. One application server cluster with a running SPECjEnterprise2010 instance is assigned to Customer A. Another application server cluster is assigned to Customer B. The database server runs separately and serves the requests of all application instances. We assume that each customer has its own independent workload and its own Service Level Agreements (SLAs). The application server clusters are heterogeneous, the cluster may consist of several different machines. Given that customer workloads vary over time and new services may be deployed on-the-fly, the system has to be reconfigured dynamically to enforce SLAs while ensuring efficient resource utilization. Some examples of dynamic reconfigurations are the addition/removal of cluster nodes to scale up/down resource allocations, or the deployment of a new cluster to serve a new customer. To ensure SLA

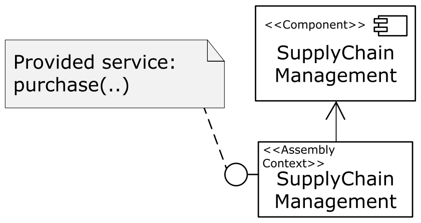


**Fig. 6.** Supply chain management component.

compliance, the service provider requires the ability to predict at run-time how application performance would be affected if the system configuration or the workload changes.

Before conducting performance predictions for all five benchmark operations in the evaluation section (Section 5), in the following we use order processing scenarios to motivate our new modeling approaches. In SPECjEnterprise2010, a dealership may trigger a manufacturing site to schedule a new work order. In case the items available in the manufacturing site's warehouse are not enough to process the order, the SCM is called to purchase additional items. Based on that user story, we describe and evaluate the new meta-model elements.

### 4.2. *Flexible service behavior abstractions*

#### 4.2.1. *Motivation*

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. The underlying performance models enabling online performance prediction must be parameterized and analyzed on-the-fly. Such models may be used in many different scenarios with different requirements for accuracy and timing constraints. Depending on the time horizon for which a prediction is made, online models may have to be solved within seconds, minutes, hours, or days. 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 parameter 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 where models at different abstraction levels would be needed. We consider the supplier domain of SPECjEnterprise2010 (see Section 4.1). 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. Fig. 6 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., [16,17]) 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 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. 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.

#### 4.2.2. *Modeling approach*

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

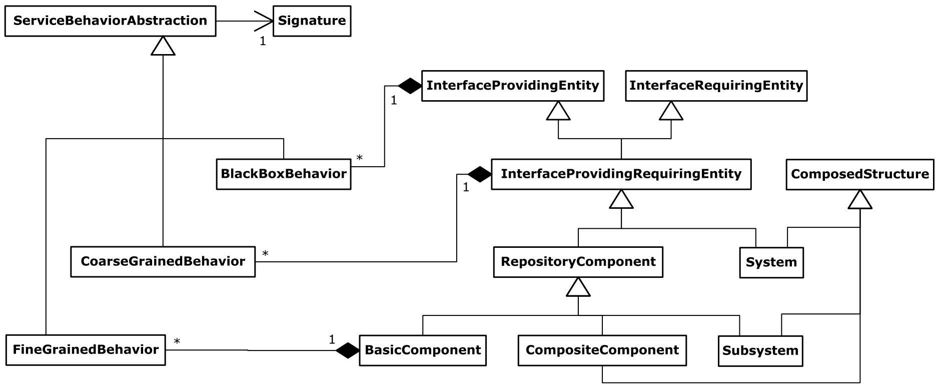


**Fig. 7.** Service behavior abstraction.

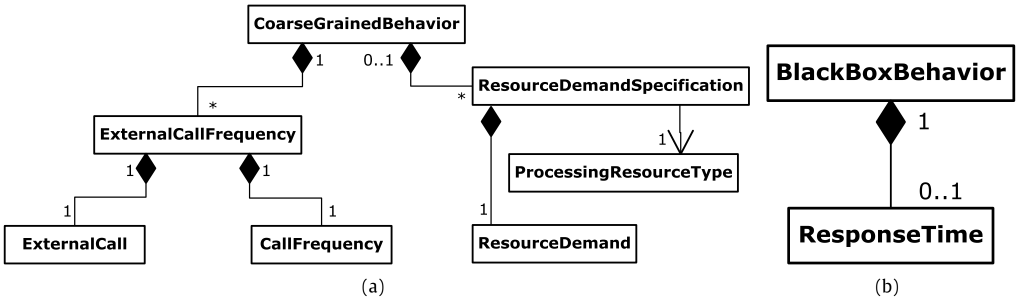


**Fig. 8.** (a) Coarse-grained and (b) black-box behavior abstractions.

*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 [6]. 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 Section 4.3. In contrast to the coarse-grained behavior description, a fine-grained behavior description requires information about the internal performance-relevant service control flow including information about the resource consumption of internal service actions.

If a service is modeled at different behavior abstraction levels, e.g., a service can be described by both a coarse-grained abstraction and a black-box behavior abstraction, we do not require a conformance relation between the different behavior descriptions. As explained in the previous section, the separate behavior descriptions may be based on different types of monitoring data.

### 4.2.3. Meta-model elements

Fig. 7 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 type `InterfaceProvidingRequiringEntity` that generalizes the types `System`, `Subsystem`, `CompositeComponent` and `BasicComponent`. Type `BlackBoxBehavior` is attached to type `InterfaceProvidingEntity`, neglecting external service calls to required services. Thus, in contrast to the fine-grained abstraction level, the coarse-grained and black-box behavior descriptions can also be attached to service-providing *composites*, i.e., `ComposedStructures`.

The meta-model elements for the `CoarseGrainedBehavior` and `BlackBoxBehavior` abstractions are shown in Fig. 8. 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.

Fig. 9 shows the meta-model elements for the fine-grained behavior abstraction. A `ComponentInternalBehavior` models the abstract control flow of a service implementation. Calls to required services are modeled using so-called

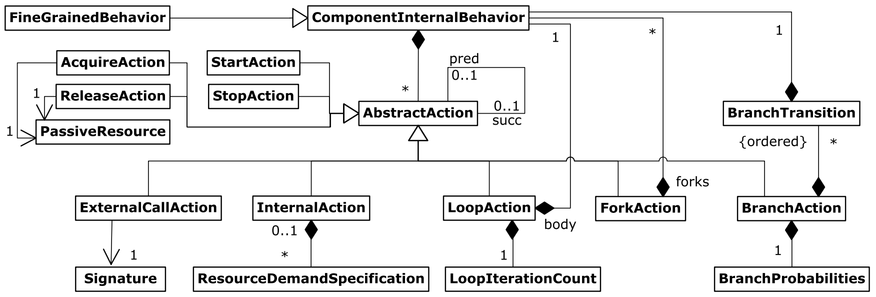


**Fig. 9.** Fine-grained behavior abstraction.

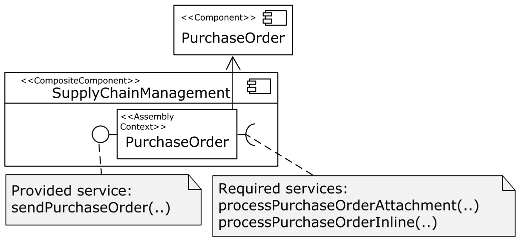


**Fig. 10.** SCM component internals.

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.

### 4.2.4. *Evaluation*

In our scenario, as shown in Fig. 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. In the benchmark application, the probability of an inline message or a message with attachment each equals to 50%.

A fine-grained model of service sendPurchaseOrder is depicted in Fig. 11. Internal service behavior is taken into account by reflecting the service's internal control flow. There is a branch action that either leads to an external service call to processPurchaseOrderAttachment or an external service call to processPurchaseOrderInline, both with a probability of 0.5.

A coarse-grained model of service sendPurchaseOrder is depicted in Fig. 12. The external service calls to processPurchaseOrderAttachment and processPurchaseOrderInline are modeled as they can be observed from the component boundary of component PurchaseOrder. For each call to sendPurchaseOrder, a respective external service is either called once or not at all. For both cases, one observes a probability of 0.5. In the model, this call frequency is described with the probability mass function IntPMF[(0;0.5)(1;0.5)]. However, the exclusive relationship between the two external service calls is not reflected in the coarse-grained model.

Fig. 13(c) shows measurements of the response time of sendPurchaseOrder as a histogram. The measurements were obtained during a benchmark run under medium load with a steady state time of 15 min. As expected, the measured response time distribution is multi-modal. We compare the measured response time distribution with predicted response time distributions using (i) the fine-granular model (Fig. 13(a)) and (ii) the coarse-grained model (Fig. 13(b)). The resource demanding behavior of external service processPurchaseOrderInline was described as exponential service time with a mean of 10 ms. Service processPurchaseOrderAttachment has a 30 ms higher resource demand.

The fine-granular model reflects the bimodal distribution of sendPurchaseOrder better than the coarse-grained model. This is because the interdependency between the two external service calls is reflected in the fine-grained model as branch action, while ignored in the coarse-grained model. The latter obviously affects the response time distribution, however, the mean values of the response time predictions do not differ. Both predictions show a mean of 26 ms which well reflects the measured response time mean of 29 ms. If one is only interested in predicting the mean response time, the coarse-grained abstraction is sufficient. For a more representative response time distribution, the fine-grained abstraction is more appropriate.

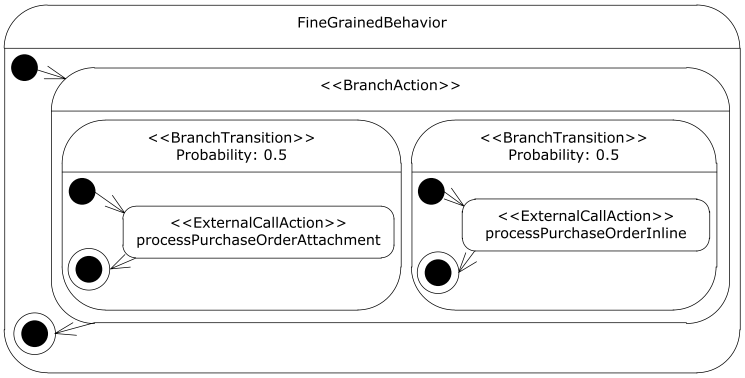

FineGrainedBehavior
<<BranchAction>>
<<BranchTransition>> Probability: 0.5
<<ExternalCallAction>> processPurchaseOrderAttachment
<<BranchTransition>> Probability: 0.5
<<ExternalCallAction>> processPurchaseOrderInline

**Fig. 11.** Fine-grained behavior abstraction of `PurchaseOrder#sendPurchaseOrder`.

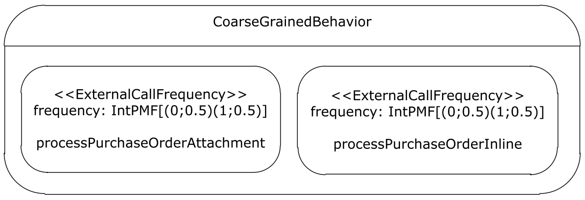

CoarseGrainedBehavior
<<ExternalCallFrequency>> frequency: IntPMF[(0;0.5)(1;0.5)] processPurchaseOrderAttachment
<<ExternalCallFrequency>> frequency: IntPMF[(0;0.5)(1;0.5)] processPurchaseOrderInline

**Fig. 12.** Coarse-grained behavior abstraction of `PurchaseOrder#sendPurchaseOrder`.

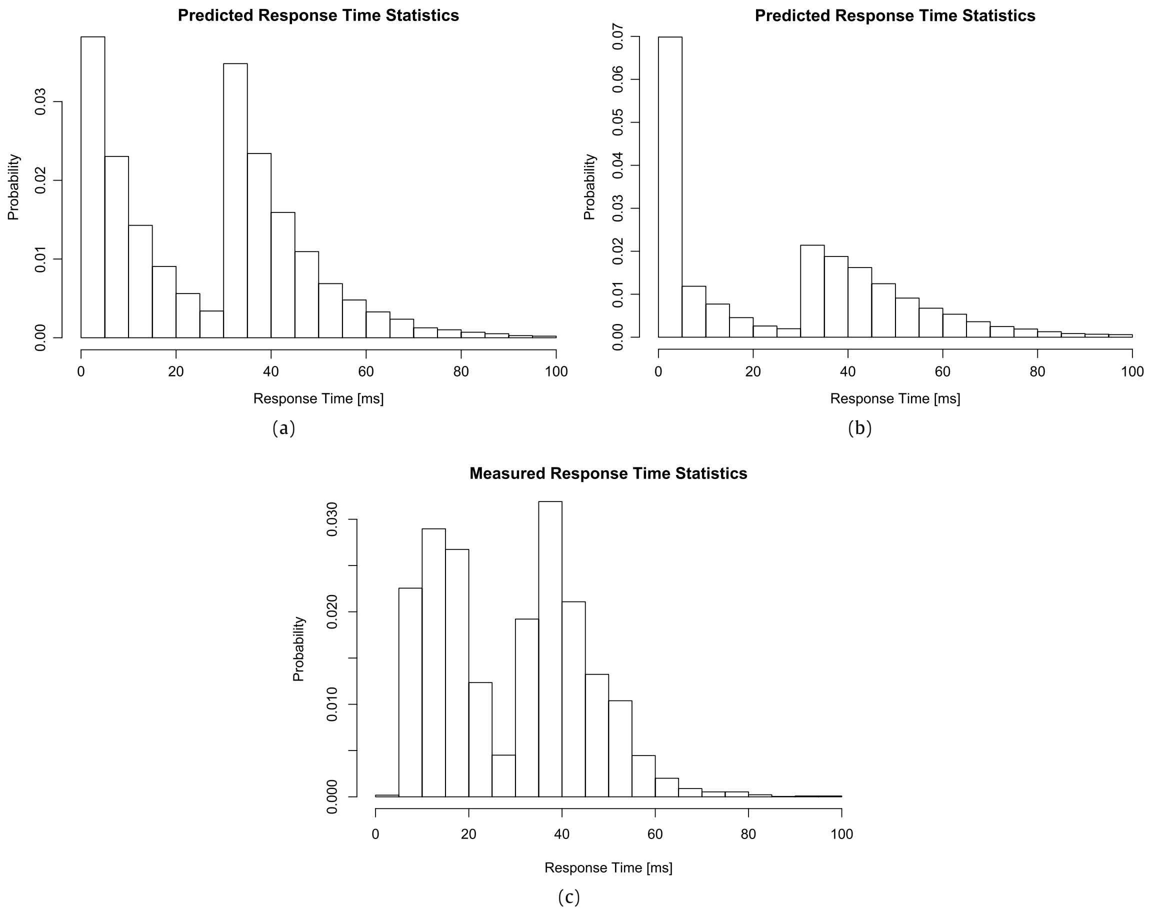


**Fig. 13.** Response time statistics of `sendPurchaseOrder`: (a) predicted using fine-grained behavior abstraction (b) predicted using coarse-grained behavior abstraction (c) measured.

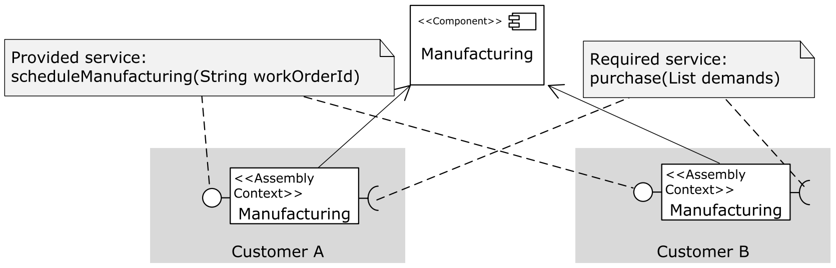


**Fig. 14.** `Manufacturing` component.

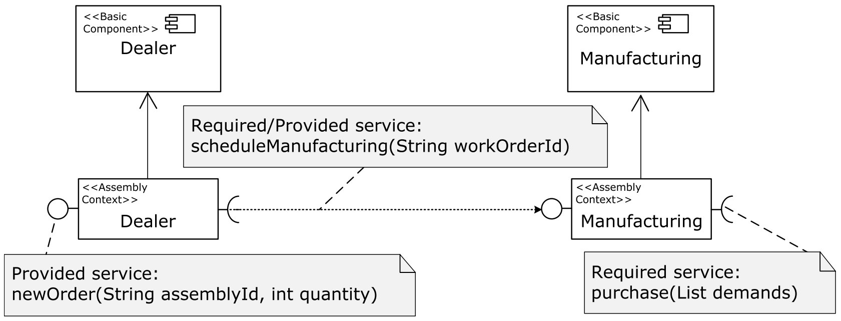


**Fig. 15.** `Dealer` and `manufacturing` components.

## 4.3. *Modeling parameter dependencies*

### 4.3.1. *Motivation*

Fig. 14 shows the `Manufacturing` component in our scenario which provides a service `scheduleManufacturing` to schedule a new work order 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. 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 3, 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. 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 Fig. 15 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 Fig. 16) 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.

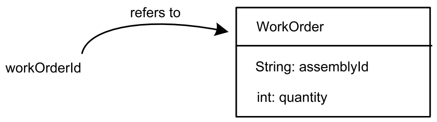


**Fig. 16.** WorkOrder data structure.

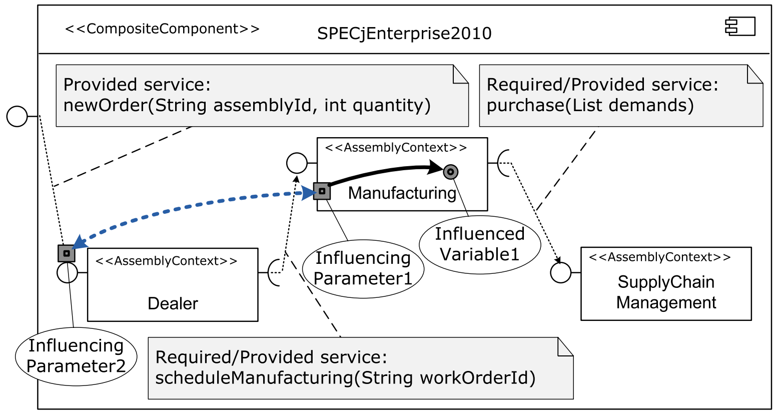


**Fig. 17.** Modeling parameter dependencies.

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. At run-time, we can monitor the dependency between the values of influencing parameter `quantity` and the observed relative frequency of the `purchase` service calls.

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. 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.

### 4.3.2. *Modeling approach*

To allow the modeling of the above described "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. Often such a mapping will only be feasible at deployment time once the complete system architecture and execution environment is available.

Fig. 17 illustrates our modeling approach in the context of the presented example from Fig. 15. The branch probability of calling the `purchase` service in the control flow of the `scheduleManufacturing` service 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 illustrated earlier (Fig. 24). 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

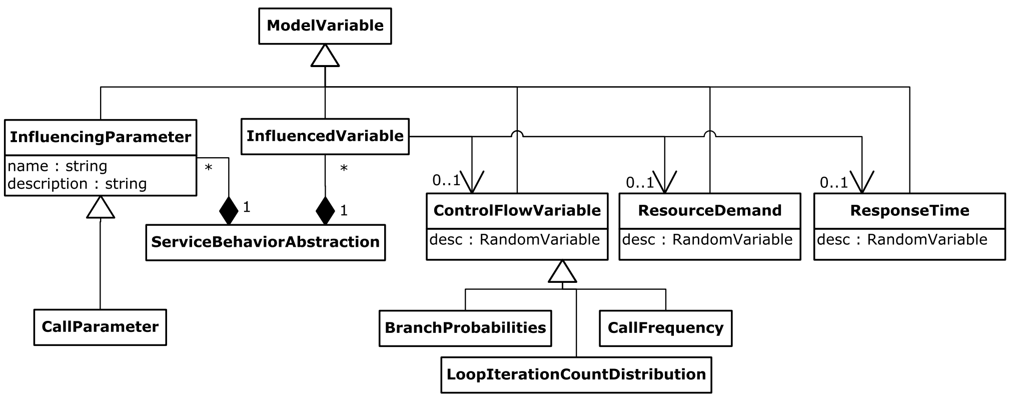


**Fig. 18.** Model variables.

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 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.

### *4.3.3. Meta-model elements*

In the following, we first describe the notion of model variables to describe both influencing parameters and influenced variables. Then we present the meta-model elements to model the different types of relationships between model variables as they are shown in Fig. 17. After presenting the semantics of the meta-model elements representing the above-mentioned scopes, we describe how we characterize the introduced relationships.

*Model variables.* The model variables involved in dependency specifications are divided into influenced variables and influencing parameters. As shown in Fig. 18, model variables that can be referenced as `InfluencedVariable` include resource demands and control flow variables (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 Fig. 19) are special types of influencing parameters. Given that in performance models, a service call parameter is only modeled if it is performance-relevant (see Fig. 20), 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 Fig. 21, we distinguish the two 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 paper, 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.

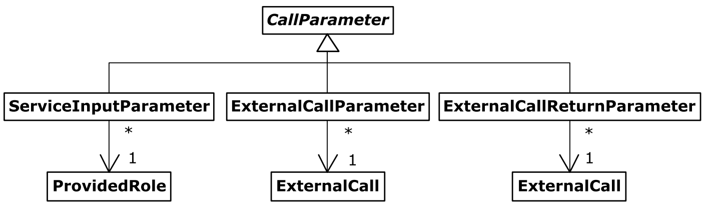


**Fig. 19.** Call parameter hierarchy.

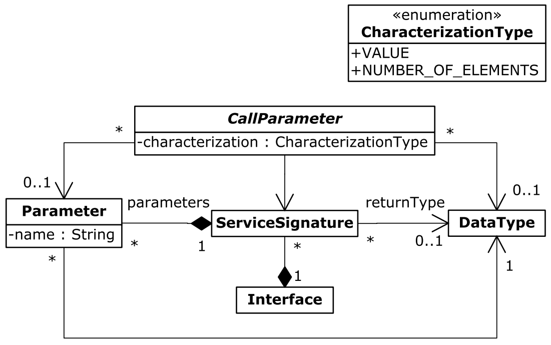


**Fig. 20.** Call parameters.

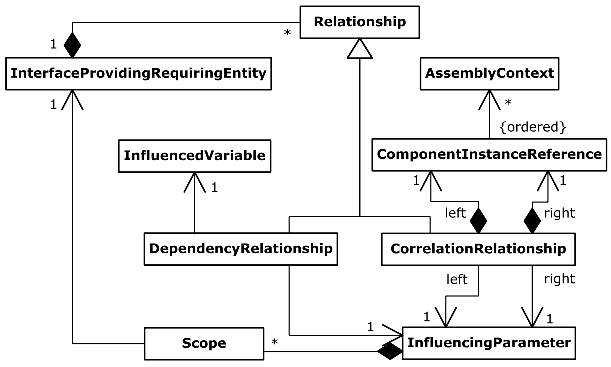


**Fig. 21.** Relationships between `InfluencedVariables` and `InfluencingParameters`.

A correlation is specified when a composed entity such as a `Subsystem` 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 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 Fig. 17 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 used to uniquely refer to a component instance are represented as ordered lists of `AssemblyContexts`.

*Scopes.* As shown in Fig. 21, we decided to model a scope as a reference to a `InterfaceProvidingRequiringEntity`. An `InfluencingParameter` may have several scopes. Let $p$ be an influencing parameter, and $C(p)$ the (composite)

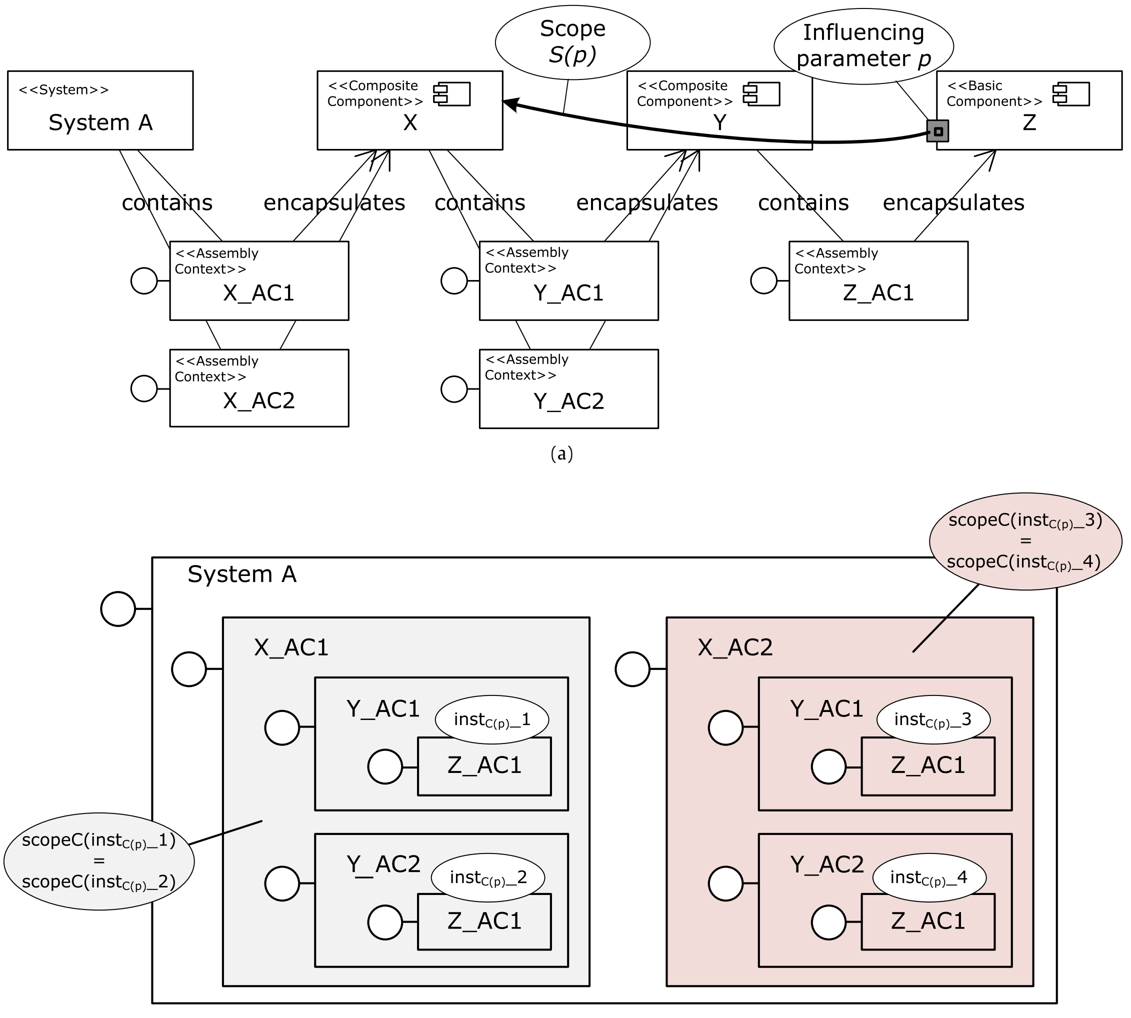


**Fig. 22.** Example of a nested composed structure: (a) showing the component types, (b) showing the composition of components.

component or (sub-)system where the influencing parameter $p$ resides. Let $S(p) = \{scopeC_1, \ldots, scopeC_n\}$ denote the set of (composite) components or (sub-)systems referenced as scopes of $p$. Set $S_0(p) = S(p) \cup \{scopeC_0\}$, where $scopeC_0$ equals to the global system. Then, for an instance of $C(p)$, denoted as $inst_{C(p)}$, the influencing parameter's scope is defined as $scopeC(inst_{C(p)}) = scopeC_i \in S_0(p)$, where $scopeC_i$ is the most inner composite in $S_0(p)$ when traversing from $inst_{C(p)}$ to the system boundary.

In Fig. 22, we provide an example illustrating the scope concept. As depicted in Fig. 22(a), component Z has an influencing parameter $p$ with a scope reference to composite component X. Fig. 22(b) shows four instances of component Z nested in composite components. The instances $inst_{C(p)}\_1$, $inst_{C(p)}\_2$ share the same scope which is different from the scope of $inst_{C(p)}\_3$ and $inst_{C(p)}\_4$. In the figure, the scopes of these instances are illustrated as light gray respectively light red box.

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 Fig. 23). 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 (Fig. 24) and can be

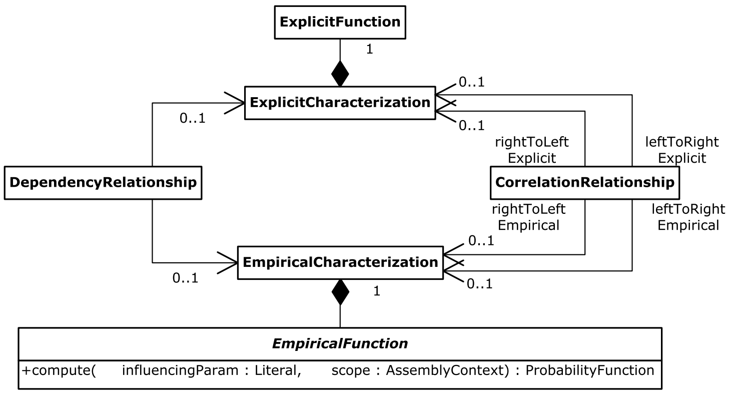


**Fig. 23.** `ExplicitCharacterization` and `EmpiricalCharacterization`.

represented by abstracting the obtained statistics as illustrated in Fig. 24(b), e.g., using equidistant or equiprobable bins. In this paper, we focus on the modeling concepts at the meta-model level. Due to lack of space, 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 $f : \texttt{Literal} \longrightarrow \texttt{ProbabilityFunction}$. It maps a value of $p$ to a probability function describing $v$. The signature of an empirical function $\hat{f}$ is $\hat{f} : \texttt{Literal} \times \texttt{AssemblyContext} \longrightarrow \texttt{ProbabilityFunction}$. 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 $inst_{C(p)}$. 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 $inst_{C(p)}$, i.e., the assembly context of $scopeC(inst_{C(p)})$.

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., $f_{leftToRight}$ and $f_{rightToLeft}$ for explicit functions, respectively $\hat{f}_{leftToRight}$ and $\hat{f}_{rightToLeft}$ for empirical functions. For the explicit functions, the scopes of the involved influencing parameters $p_{left}$ and $p_{right}$ are ignored. For the empirical functions, the intersection of the scopes $p_{left}$ and $p_{right}$ has to be considered.

### *4.3.4. Evaluation*

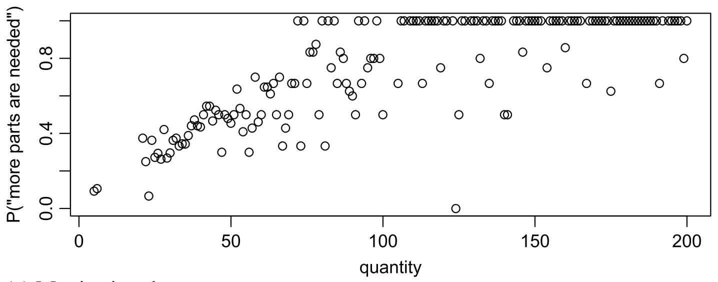
In Section 4.3.1, we described a parameter dependency between the values of parameter `quantity` and the probability of calling the `purchase` service (see Fig. 15). In the following, we use this parameter dependency in several evaluation scenarios.

Fig. 24(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 called. Fig. 24(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.

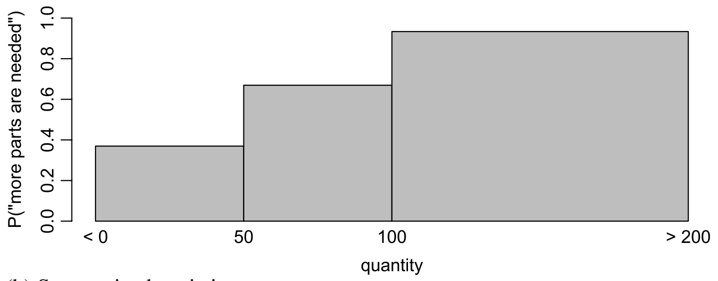
To further illustrate the relevance of the considered parameter dependency, we conducted experiments with the `scheduleManufacturing` workload. In the first scenario, we called `scheduleManufacturing` varying the quantity parameter at random in a range between 0 and 10. For the second scenario, we varied the quantity parameter between 0 and 200, while keeping the workload intensity at the same level as in the first scenario. The measurements were obtained in experiment runs with a steady state time of 15 min. Knowing the monitoring statistics in Fig. 24(a) we expect the number of `purchase` calls in the second scenario to be higher than in the first scenario. Given that `scheduleManufacturing` calls the `purchase` service asynchronously using point-to-point messaging provided by the Java Message Service (JMS), the more frequent `purchase` calls in the second experiment should not affect the response time of `scheduleManufacturing` directly but result in a higher application server utilization.

The results of the experiment runs are shown in Fig. 25. While in the first experiment ("quantity in [0..10]") the average utilization of the Oracle WebLogic Server (U_WLS) is approximately 17%, it increases to 63% in the second experiment ("quantity in [0..200]"). The increased mean response time (R_avg) of service `scheduleManufacturing` can be explained by the increased utilization.

Using a fine-grained behavior model, with a characterization of the parameter dependency as shown in Fig. 24(b), we made predictions for both scenarios. Regarding the second scenario, the utilization U_WLS and average response time R_avg differ from the actual measurements with a small, negligible error. However, concerning the first scenario, with



(a) Monitoring data.



(b) Summarized statistics.

**Fig. 24.** `scheduleManufacturing` statistics.

| Workload: `scheduleManufacturing` | Measurements | | Predictions | |
|---|---|---|---|---|
| | U_WLS | R_avg [ms] | U_WLS | R_avg [ms] |
| quantity in [0..10] | 0.166 | 17.2 | 0.365 | 19.4 |
| quantity in [0..200] | 0.627 | 32.9 | 0.607 | 28.3 |

**Fig. 25.** Measurements and predictions of `scheduleManufacturing` scenarios.

U_WLS = 0.365 the prediction particularly overestimates the server utilization. The overestimation is the result of the summarized monitoring statistics. For the interval between 0 and 50, the model simplifies the probability of a `purchase` call to be 0.37. Adapting the model by mapping the interval between 0 and 10 as suggested by Fig. 24(b) to a `purchase` call probability of, e.g., 0.10, the prediction yields representative results: With U_WLS = 0.196 and R_avg = 16.8 ms, the predictions match the measurements also in the first scenario.

The evaluation shows the performance-relevance of the parameter dependency between the influencing parameter `quantity` and the observed relative frequency of the `purchase` service calls. It further shows that detailed monitoring statistics may be of importance for a representative performance model. 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 in Fig. 24(a) 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 a so-called 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., [17]. We employed monitor throttling strategies 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 WLS 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 can be configured taking the trade-off between overhead and monitoring information into account.

### *4.4. Resource landscape*

#### *4.4.1. Motivation*

Having modeled the performance-relevant behavior of the software architecture as described in Sections 4.2 and 4.3, predicting the performance requires information about the execution environment (the resource landscape) where the software is deployed.

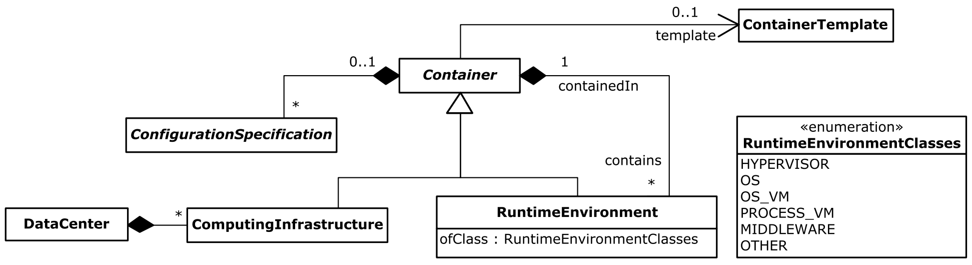


**Fig. 26.** Hierarchical run-time environments in the resource landscape.

Today's IT execution environment implement abstraction layers like virtualization and middleware technologies. This promises to be beneficial when reacting on changes of service workloads and resource consumption by provisioning and allocating resources as they are needed. However, introducing new abstraction layers to increase flexibility comes at the cost of increased system complexity. The new abstraction layers provide new reconfiguration possibilities that have an impact on the system performance. Existing model-based resource management and capacity planning techniques such as [6,18] 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. For instance, the virtualization overhead [19] affecting resource utilization and service response times is typically ignored.

### *4.4.2. Modeling approach*

Current techniques do not provide means to model the layers of the component execution environment (e.g., the virtualization layer) explicitly. The performance influences of the individual layers, the dependencies among them and the resource allocations at each layer should be captured as part of the models. This is necessary in order to be able to predict at run-time how a change in the execution environment (e.g., modifying resource allocations at the virtual machine level) would affect the system performance. The resource landscape meta-model should reflect the static view of a distributed execution environment, i.e., provide 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 machines, virtual CPUs, etc.

An instance of the resource landscape model can then be annotated with dynamic aspects. In [12], we introduced the adaptation points sub-meta-model of DMM that can be used to describe which parts of the DMM resource landscape model instance are variable and can be adapted during operation, i.e., it provides possibilities to specify the configuration range of the dynamic system. Note that the adaptation points meta-model [12] as well as the adaptation language we presented in [13] are out of scope of this paper.

### *4.4.3. Meta-model elements*

In this section, we show only a part of the DMM resource landscape meta-model. A detailed description is provided in [12] or in the technical report [11]. Fig. 26 depicts the part of the meta-model describing the nested containment principle to model resource landscapes. For example, imagine servers which 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 at run-time (e.g., when a VM is migrated). The central entity of this meta-model is the abstract entity `Container`. We distinguish between two major types of containers, the `ComputingInfrastructure` and the `RuntimeEnvironment`. A `ComputingInfrastructure` forms the root element in our tree-like structure of containers and corresponds to a physical machine. It cannot be contained by another container but it can have nested containers. The `RuntimeEnvironment` is a generic model element to build nested layers, i.e., it can be contained within a container and it might contain further run-time environments. Each `RuntimeEnvironment` has a property to specify the class of the `RuntimeEnvironment`. These classes are listed in the enumeration type `RuntimeEnvironmentClasses`. The classes are `HYPERVISOR` for the different hypervisors of virtualization platforms, `OS` for operating systems, `OS_VM` for virtual machines emulating standard hardware, `PROCESS_VM` for virtual machines like the Java VM, `MIDDLEWARE` for middleware environments, and `OTHER` for any other type. The consistency within the modeled layers is ensured with Object Constraint Language (OCL) expressions. The implemented constraints prohibit the instantiation of different `RuntimeEnvironment` classes within one container.

Furthermore, the type `Container` has a property to specify its configuration. We distinguish three different types of configuration specifications: `ActiveResourceSpecification`, `PassiveResourceSpecification`, and `CustomConfigurationSpecification` (see Fig. 27(a)). The purpose of the `ActiveResourceSpecification` is to specify the active resources a `Container` provides. Currently supported `ProcessingResourceTypes` are CPU and HDD. Network resources are out of scope of this paper. Using the properties `schedulingPolicy`, `processingRate` and `numberOfParallelProcessingUnits`, for example, a dualcore CPU can be specified with `PROCESSOR_SHARING` as `schedulingPolicy`, a `processingRate` of 2.4 GHz and a `numberOfParallelProcessingUnits` of 2. The `PassiveResourceSpecification` can be used to specify properties of passive resources. Passive resources can be,

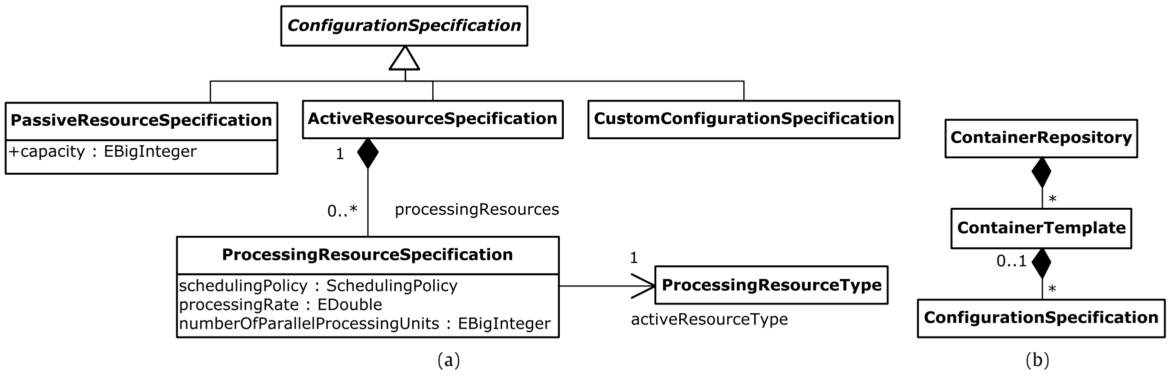

ConfigurationSpecification
PassiveResourceSpecification
+capacity : EBigInteger
ActiveResourceSpecification
CustomConfigurationSpecification
1
0..*
processingResources
ProcessingResourceSpecification
schedulingPolicy : SchedulingPolicy
processingRate : EDouble
numberOfParallelProcessingUnits : EBigInteger
1
ProcessingResourceType
activeResourceType
(a)
ContainerRepository
*
ContainerTemplate
0..1
*
ConfigurationSpecification
(b)

**Fig. 27.** (a) Configuration specification and (b) container template repository.

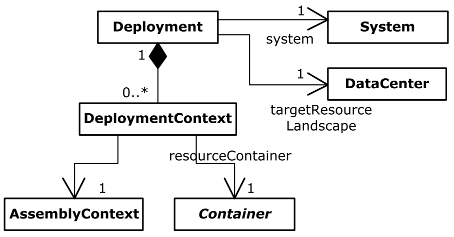


**Fig. 28.** Deployment.

e.g., the main memory size or software resources such as database connection thread pools. The attribute `capacity` is used to specify, e.g., the number of threads. In case a `Container` has an individual configuration which cannot be modeled with the previously introduced elements, one can use the `CustomConfigurationSpecification`. The semantics of this entity is described in [12].

With the meta-model concepts presented so far, it would be necessary to model each container and its configuration explicitly. This can be very cumbersome, especially when modeling clusters of several identical machines. Our 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. The concept is implemented using the `ContainerTemplate` entity. Configured container templates are collected in the `ContainerRepository` (see Fig. 27(b)). The `ContainerTemplate` is similar to a `Container` since it also may refer to `ConfigurationSpecification`. A `Container` in a resource landscape model instance might have a reference to a `ContainerTemplate` (see Fig. 26). 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. For example, assume that a container instance has no individual properties and only a reference to a template. Then, only the configuration specification of the template would be considered. However, if the container instance has an individual configuration specification, then these settings would override the properties of the template. The template mechanism is useful is situations where, e.g., a VM is cloned (use template as configuration) and later changed (individual configuration options possible).

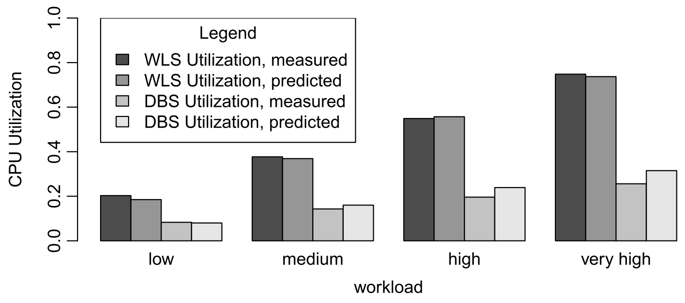
After describing the execution environment, one must specify which services are executed on which part of the resource landscape. We refer to this specification as *deployment* captured in the deployment model shown in Fig. 28 which is based on PCM [6]. A `Deployment` connects a model instance of the software architecture (`System`) with a model instance of a resource landscape (`DataCenter`). It consists of several deployment contexts that map an `AssemblyContext` contained in the system model to a `Container` that is described in the resource landscape.

### 4.4.4. *Evaluation*

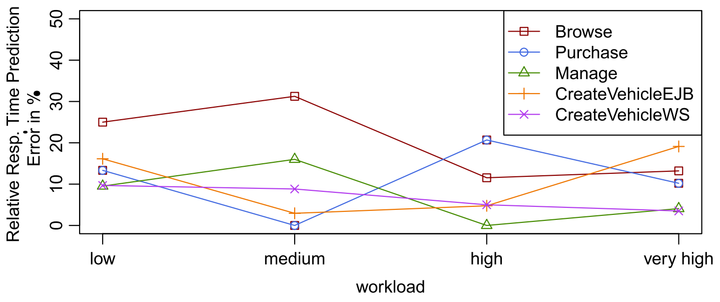
For the evaluation of the resource landscape meta-model, we refer to the work we presented in [12]. Modeling the performance impact of virtualization layers is beyond the scope of this paper. Thus, the presented measurement experiments do not involve OS virtualization layers.

## 5. Summary of evaluation

To evaluate the feasibility and effectiveness of the entire modeling approach, we deployed the benchmark in the system environment depicted in Fig. 5 in two application server clusters consisting of two and four nodes, respectively. We used PCM as a basis adopting the modeling approach presented in this paper. The resource demands were estimated based on



(a) Utlization measurements and predictions.



(b) Relative error of responce time predictions.

**Fig. 29.** Exemplary measurements and prediction results with SPECjEnterprise2010.

utilization and throughput data. As performance metrics, we considered the average response times of the five benchmark operations as well as the average CPU utilization of the WebLogic application servers (WLS CPU) and the database server (DBS CPU). We predicted the performance metrics for low load conditions (≈20% WLS CPU utilization), medium load conditions (≈40%), high load conditions (≈60%) and very high load conditions (≈80%) and then compared them with steady-state measurements under the same load conditions on the real system. Note that the response times of the benchmark operations are measured at the benchmark driver agents. We measured the delay for establishing a connection to the WLS instance which is dependent on the system load. With the knowledge of the number of connections the individual benchmark operations trigger, the load-dependent delay is considered in the predicted operation response times.

The response times of the benchmark operations vary from 10 to 70 ms. We considered a number of different scenarios varying the application workload and system configuration. The measured and predicted server utilization for the different load levels are depicted in Fig. 29(a). The utilization predictions fit the measurements very well, both for the WLS instances as well as for the DBS. Fig. 29(b) shows the relative error of the response time predictions. The error is mostly below 20%, only predictions of operation *Browse* show a higher error (30%). The prediction accuracy of the former increases with the load. This is because Browse has a rather small resource demand but includes a high number of round trips to the server (15 on average) translating in connection delays.

## 6. Related work

First we present an overview of existing performance modeling approaches, distinguishing between *predictive* performance models that capture the temporal system behavior and *architecture-level* performance models that provide means to model the performance-relevant aspects at the software architecture level. Then we refer to online performance analysis and name existing approaches on online QoS and resource management.

A survey of model-based performance prediction techniques was published in [20]. A number of techniques utilizing a range of different performance models have been proposed including product-form queueing networks (e.g., [2]), layered queueing networks (e.g., [4]), queueing Petri nets (e.g., [18]), stochastic process algebras [21], statistical regression models (e.g., [22]) and learning-based approaches (e.g., [23]). Such models capture the temporal system behavior and can be used for performance prediction by means of analytical or simulation techniques. However, these predictive performance models are normally used as high-level system performance abstractions and as such they do not explicitly distinguish the degrees-of-freedom and performance-influencing factors of the system's software architecture and execution environment. They are high-level in the sense that: (i) complex services are modeled as black boxes without explicitly capturing their internal behavior and the influences of their deployment context, configuration settings and input parameters, and (ii) the execution environment is abstracted as a set of logical resources (e.g., CPU, storage, network) without explicitly distinguishing the performance influences of the various layers (e.g., physical infrastructure, virtualization and middleware) and their configuration.

Architecture-level 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. A number of architecture-level performance meta-models have been developed by the performance engineering community in the last years. The most prominent examples are the UML SPT profile [24] and its successor the UML MARTE profile [9], both of which are extensions of UML as the de facto standard modeling language for software architectures. Other proposed meta-models include CSM [25], PCM [6], SPE-MM [8], and KLAPER [7]. A recent survey of model-based performance modeling techniques for component-based systems was published in [10].

Approaches that explicitly consider the influence of parameters in their performance analysis include [6,26–28]. The most advanced approaches concerning parametric dependencies are [6,26]. Components and their behavior can be specified in a parameterized way, considering the dependency of input and deployment specific parameters to the component's resource demand, control flow, etc. The approaches in [28,27] have less expressiveness. For example, they do not consider service input or output parameters or limit the set of parameters to, e.g., thread pool size or to resource demand parameterization. Orthogonal approaches tackling the challenge of parametric dependencies in performance analysis are [29,30]. They model the component's internal state which might lead to state space explosion. Furthermore, the approaches do not differentiate the execution time on different resources (CPU, HDD, etc.) or miss the specification of required interfaces.

The existing architecture-level performance models are normally intended for use in an offline setting at design and deployment time to evaluate alternative system designs and/or to predict the system performance for capacity planning purposes. However, as we described in Section 2, at run-time there are different requirements on the underlying performance abstractions of the system architecture and the respective performance prediction techniques.

Approaches on run-time QoS and resource management based on online performance analysis are presented in, e.g., [4,5,31]. Problem of all the approaches is that their performance analysis are rather restricted (if any) in terms of the level of detail. The main drawbacks are that they are either limited to a subset of parameters, try to abstract from important details, or do not capture important parametric dependencies.

## 7. Concluding remarks

We analyzed typical online performance prediction scenarios and the requirements on online performance-modeling approaches at the architecture-level. The analysis revealed that current architecture-level performance modeling approaches are not suitable at run-time.

We proposed new software architecture-level performance abstractions, specifically designed for use in *online* scenarios. This involves (i) a novel approach to model the component context and parameter dependencies specifically for use at run-time, (ii) a new approach to model performance-relevant service behavior at different levels of detail, and (iii) a meta-model to describe the resource environment and the deployment of the software system. We conducted a detailed evaluation of the suitability of the proposed modeling approach in the context of the SPECjEnterprise2010 benchmark providing a set of realistic and representative application scenarios.

The presented modeling abstractions are subsets of the Descartes Meta-Model (DMM) [11], a new meta-model for run-time QoS and resource management in virtualized service infrastructures. The DMM approach directly aims at the identified gaps between run-time and design-time modeling as presented in Section 2: The approach to model parameter dependencies and the flexible service behavior abstractions are designed (i) with regard to the different types and amount of data available for model parameterization and calibration as well as (ii) with regard to the trade-off between prediction accuracy and overhead. The DMM resource landscape model on the other hand allows to structure the model aligned with the system layers. The other parts of DMM for modeling the adaptation points of a running system [12] and for modeling the corresponding adaptation activities [13] allow to model the degrees-of-freedom when considering dynamic system changes at run-time such as changing workloads or resource allocations.

## References

[1] S. Kounev, F. Brosig, N. Huber, R. Reussner, Towards self-aware performance and resource management in modern service-oriented systems, in: Proceedings of the 7th IEEE International Conference on Services Computing (SCC 2010), July 5–10, IEEE Computer Society, Miami, Florida, USA, 2010.
[2] D.A. Menascé, H. Gomaa, A method for design and performance modeling of client/server systems, IEEE Transactions on Software Engineering 26 (2000) 1066–1085.
[3] R. Nou, S. Kounev, F. Julia, J. Torres, Autonomic QoS control in enterprise grid environments using online simulation, Journal of Systems and Software 82 (2009).
[4] J. Li, J. Chinneck, M. Woodside, M. Litoiu, G. Iszlai, Performance model driven qos guarantees and optimization in clouds, in: Software Engineering Challenges of Cloud Computing, 2009. CLOUD '09. ICSE Workshop on, pp. 15–22.
[5] G. Jung, M. Hiltunen, K. Joshi, R. Schlichting, C. Pu, Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures, in: Distributed Computing Systems, ICDCS, 2010 IEEE 30th International Conference on, pp. 62–73.
[6] S. Becker, H. Koziolek, R. Reussner, The palladio component model for model-driven performance prediction, Journal of Systems and Software 82 (2009) 3–22.
[7] V. Grassi, R. Mirandola, A. Sabetta, Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach, Journal of Systems and Software 80 (2007) 528–558.

[8] C. U. Smith, C. M. Lladó, V. Cortellessa, A. Di Marco, L. G. Williams, From UML models to software performance results: An SPE process based on XML interchange formats, in: WOSP '05: Proceedings of the 5th International Workshop on Software and Performance, ACM Press, New York, NY, USA, 2005, pp. 87–98.
[9] Object Management Group (OMG), UML profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE), 2006.
[10] H. Koziolek, Performance evaluation of component-based software systems: A survey, Performance Evaluation (2009).
[11] S. Kounev, F. Brosig, N. Huber, Descartes Meta-Model (DMM), Technical Report, Karlsruhe Institute of Technology (KIT), 2012, http://www.descartes-research.net/metamodel/.
[12] N. Huber, F. Brosig, S. Kounev, Modeling dynamic virtualized resource landscapes, in: Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures, QoSA '12, ACM, New York, NY, USA, 2012, pp. 81–90.
[13] N. Huber, A. van Hoorn, A. Koziolek, F. Brosig, S. Kounev, S/T/A: Meta-modeling run-time adaptation in component-based system architectures, in: 9th IEEE International Conference on e-Business Engineering, ICEBE 2012, Hangzhou, China.
[14] F. Brosig, N. Huber, S. Kounev, Modeling parameter and context dependencies in online architecture-level performance models, in: Proceedings of the 15th ACM SIGSOFT International Symposium on Component Based Software Engineering, CBSE 2012, June 26–28, 2012, Bertinoro, Italy.
[15] SPECjEnterprise2010 Design Document, http://www.spec.org/jEnterprise2010/docs/DesignDocumentation.html, 2011.
[16] G. Pacifici, W. Segmuller, M. Spreitzer, A.N. Tantawi, CPU demand for web serving: Measurement analysis and dynamic estimation, Performance Evaluation (2008).
[17] F. Brosig, N. Huber, S. Kounev, Automated extraction of architecture-level performance models of distributed component-based systems, in: 26th Int. Conf. on Automated Software Engineering, ASE'11.
[18] S. Kounev, Performance modeling and evaluation of distributed component-based systems using queueing Petri nets, IEEE Transactions on Software Engineering 32 (2006) 486–502.
[19] N. Huber, M. von Quast, M. Hauck, S. Kounev, Evaluating and modeling virtualization performance overhead for cloud environments, in: Proceedings of the 1st International Conference on Cloud Computing and Services Science (CLOSER 2011), SciTePress, Noordwijkerhout, The Netherlands, 2011, pp. 563–573.
[20] S. Balsamo, A. Di Marco, P. Inverardi, M. Simeoni, Model-based performance prediction in software development: A survey, IEEE Transactions on Software Engineering 30 (2004).
[21] S. Gilmore, V. Haenel, L. Kloul, M. Maidl, Choreographing security and performance analysis for web services, in: M. Bravetti, L. Kloul, G. Zavattaro (Eds.), Formal Techniques for Computer Systems and Business Processes, in: Lecture Notes in Computer Science, vol. 3670, Springer, Berlin, Heidelberg, 2005, pp. 200–214.
[22] E. Eskenazi, A. Fioukov, D. Hammer, Performance prediction for component compositions, Proceedings of the 7th International Symposium on Component-based Software Engineering (CBSE).
[23] A. Elkhodary, N. Esfahani, S. Malek, Fusion: A framework for engineering self-tuning self-adaptive software systems, in: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, ACM, New York, NY, USA, 2010, pp. 7–16.
[24] Object Management Group (OMG), UML-SPT: UML Profile for Schedulability, Performance, and Time, v1.1, 2005.
[25] D. Petriu, M. Woodside, An intermediate metamodel with scenarios and resources for generating performance models from UML designs, Software and Systems Modeling 6 (2007).
[26] E. Bondarev, J. Muskens, P. d. With, M. Chaudron, J. Lukkien, Predicting real-time properties of component assemblies: A scenario-simulation approach, in: EUROMICRO, pp. 40–47.
[27] X. Wu, M. Woodside, Performance modeling from software components, in: WOSP '04: Proceedings of the Fourth International Workshop on Software and Performance, vol. 29, ACM Press, New York, NY, USA, 2004, pp. 290–301.
[28] A. Bertolino, R. Mirandola, CB-SPE Tool: Putting component-based performance engineering into practice, in: Proceedings of the 7th International Symposium on Component-Based Software Engineering, CBSE, in: LNCS, vol. 3054, Edinburgh, UK, 2004, pp. 233–248.
[29] D. Hamlet, Tools and experiments supporting a testing-based theory of component composition, ACM Transactions on Software Engineering and Methodology 18 (2009) 12:1–12:41.
[30] M. Sitaraman, G. Kulczycki, J. Krone, W. F. Ogden, A. L. N. Reddy, Performance specification of software components, SIGSOFT Software Engineering Notes 26 (2001) 3–10.
[31] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, Automated control of multiple virtualized resources, in: Proceedings of the 4th ACM European Conference on Computer Systems, EuroSys '09, ACM, New York, NY, USA, 2009, pp. 13–26.