Modeling Parameter and Context Dependencies in Online Architecture-Level Performance Models*

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

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

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

ABSTRACT

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 dynamic changes in the system environment and adapt the system configuration accordingly. Architecture-level performance models provide a powerful tool for performance prediction, however, current approaches to modeling the execution context of software components are not suitable for use at run-time. In this paper, we analyze the typical online performance prediction scenarios and propose a novel performance meta-model for expressing and resolving parameter and context dependencies, specifically designed for use in *online* scenarios. We motivate and validate our approach in the context of a realistic and representative online performance prediction scenario based on the SPECjEnterprise2010 standard benchmark.

Categories and Subject Descriptors

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

Keywords

Architecture-level performance model, Parameter dependencies

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 technologies like virtualization, or 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 dynamic changes in the system environment and predict their effect on the system QoS. Service providers are often faced with questions such as: What performance would a new service deployed on the virtualized infrastructure exhibit and how much resources should be allocated to it? How should the system configuration be adapted to avoid performance issues or inefficient resource usage arising from changing customer workloads? What would be the effect of changing resource allocations and/or migrating a service from one physical server to another? Answering such questions requires the ability to predict at *run-time* how the performance of running services would be affected if the system configuration or the workload changes. We refer to this as *online performance prediction* [12].

Existing approaches to online performance prediction (e.g., [15, 16, 14, 10]) 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 and configuration. Services are typically modeled as black boxes and many restrictive assumptions are often imposed. 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., [1, 6, 21, 17]). 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* [13] 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.

This paper argues that current approaches to modeling the component execution context in architecture-level per-

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

CBSE'12, June 26–28, 2012, Bertinoro, Italy.

formance 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. Component performance models typically have multiple parameters such as usage profile parameters, resource demand parameters, and control flow parameters. We argue that there are some fundamental differences in the type and amount of data available as a basis for model parameterization and calibration at system design-time vs. run-time.

At system design-time, model parameters are often estimated based on approximation techniques or measurements if implementations of the system components exist. In a controlled testing environment, theoretically one could conduct arbitrary experiments under different settings to evaluate parameter dependencies. However, in practice, this possibility typically cannot be exploited adequately, due to the inavailability of a realistic production-like execution environment and/or the lack of complete implementations of all system components. Thus, model parameters are often approximated very roughly based on experiments in a small testing environment.

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 and possibly complex interactions between software components only observable at run-time. However, during operation, we don't have the possibility to run arbitrary experiments since the system is in production. In such a setting, monitoring has to be done with care, keeping the monitoring overhead within limits such that the system operation is not disturbed.

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 approach for modeling the component execution context and parameter dependencies specifically designed for use at run-time. We use the Palladio Component Model (PCM) [1] as a basis given that it is one of the most advanced component-based performance meta-models in terms of parametrization and tool support. We motivate and validate our approach 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 the 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) detailed evaluation of the suitability of the proposed modeling approach 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. The presented modeling abstractions are an integral part 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 modeling abstractions for the application architecture level. Other parts of DMM describe the service deployment using a resource environment model that reflects, e.g., the physical infrastructure and the virtualization layer (see [9]).

The remainder of this paper is organized as follows. Section 2 discusses current approaches to modeling parameter and context dependencies in performance models as implemented in PCM. In Section 3, we present our proposed 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 4. Finally, we review related work in Section 5 and wrap up in Section 6.

2. BACKGROUND

One of the most advanced architecture-level performance modeling languages, in terms of parametrization and tool support, is the Palladio Component Model (PCM) [1]. 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. 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

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

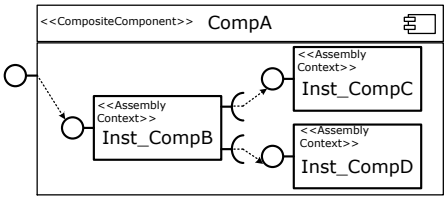

Figure 1: Assembly of Composite Component

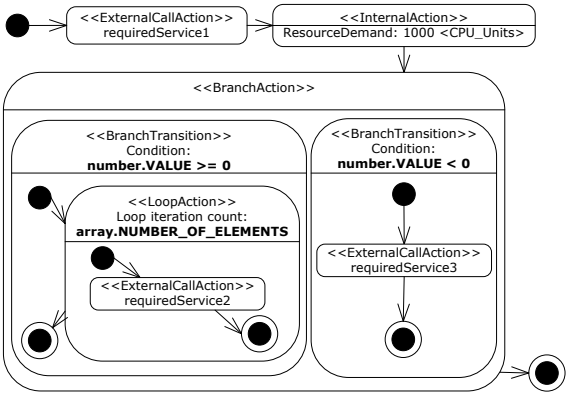

Figure 2: RDSEFF of Service with Signature `execute(int number, List array)` (cf. [1])

interface. An example of how a composite component is assembled is shown in Figure 1. Component `CompA` comprises three instances of basic components connected according to their provided and required interfaces.

Service Behavior Abstraction. For each service a component provides, in PCM the service's internal behavior is modeled using a Resource Demanding Service Effect Specification (RDSEFF) [1]. 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)`[1] is shown in Figure 2. 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.

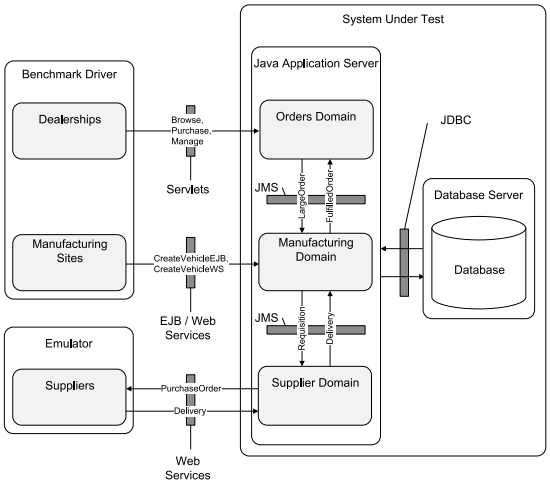

Figure 3: SPECjEnterprise2010 Architecture [22]

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.

3. PERFORMANCE META-MODEL

In this section, we present a new performance meta-model for expressing and resolving parameter and context dependencies specifically designed for use in online scenarios. We use PCM as a basis motivating and validating our approach in the context of a realistic and representative online performance prediction scenario based on the industry-standard SPECjEnterprise2010 benchmark.

3.1 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.

Figure 3 depicts the architecture of the benchmark. We refer the reader to [22] for further details on the benchmark. We consider a scenario where a set of customers are running their applications in a virtualized data center infrastructure. In our case, each customer is running an instance of SPEC-

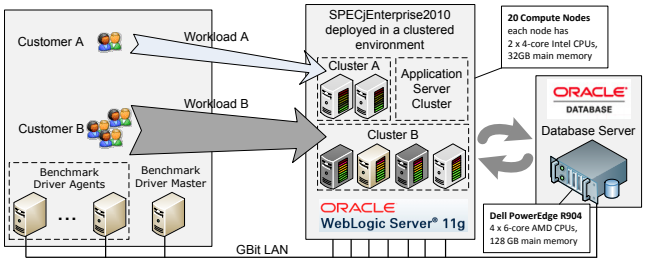

Figure 4: SPECjEnterprise2010 Deployment

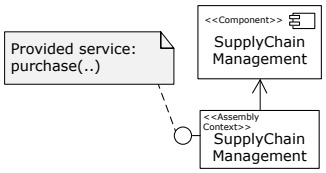

Figure 5: Supply Chain Management Component

jEnterprise2010 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 Figure 4.

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 physical or virtual 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 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.

3.2 Flexible Service Behavior Abstractions

3.2.1 Motivation

In the described scenario, 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' 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 Sect. 3.1). 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. Figure 5 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 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., [18, 4]) as well as frequencies of calls to other components. Thus, in this case, a more fine granular model of the component can be built, allowing to predict its response time and resource utilization for different workloads.

Finally, if the internal behavior of the `SCM` component including its control flow and resource consumption of internal actions can be monitored, more detailed models can be built allowing to obtain more accurate performance predictions including response time distributions. Predictions of response time distributions are relevant for example when evaluating SLAs with service response time limits defined in terms of response time percentiles. In our scenario, as shown in Figure 6, the `SCM` component is implemented as a composite component containing a child component `PurchaseOrder`. The latter is responsible for dispatching the purchase orders (service `sendPurchaseOrder`). The sending operation supports two modes of operation: i) sending the order as an inline message without attachments, or ii) sending the order as a message with attachment. Figure 7 shows some measurements of the response time of `sendPurchaseOrder` as a histogram. As expected, the measured response time distribution is multi-modal. Thus, to predict the response time distribution of the `sendPurchaseOrder` opera-

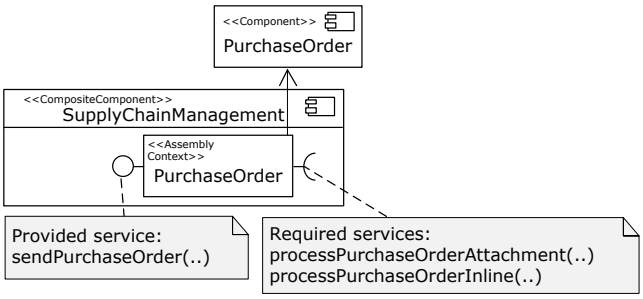

Figure 6: SCM Component Internals

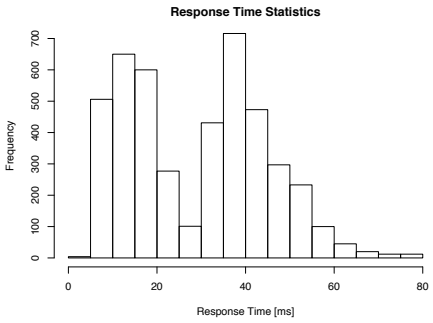

Figure 7: Response Time Statistics of sendPO

tion, a fine-granular model of its internal behavior is needed taking into account its internal control flow.

3.2.2 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:

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 [1]. The control flow is modeled at the same abstraction level as in PCM, however, our approach has some significant differences in the way model variables and parameter dependencies are modeled. The details of these are presented in detail in Section 3.3. 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.

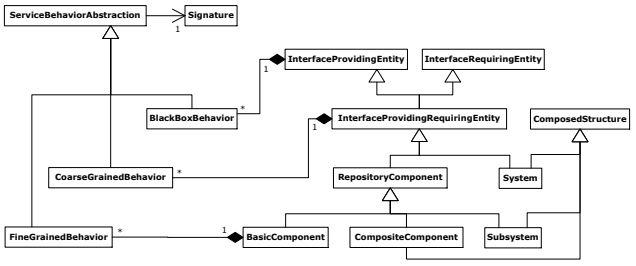

Figure 8: Service Behavior Abstraction

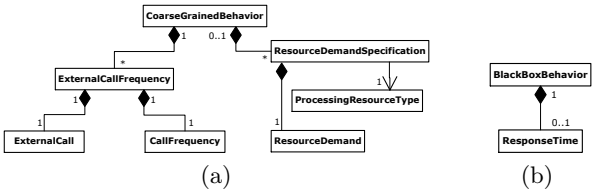

Figure 9: (a) Coarse-Grained and (b) Black-Box Behavior Abstractions

3.2.3 Meta-model

Figure 8 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 Figure 9. A `CoarseGrainedBehavior` consists of `ExternalCallFrequencies` and `ResourceDemandSpecifications`. An `ExternalCallFrequency` characterizes the type and the number of external service calls. Type `ResourceDemandSpecification` captures the total service time required from a given `ProcessingResourceType`. A `BlackBoxBehavior`, on the other hand, can be described with a `ResponseTime` characterization.

Figure 10 shows the meta-model elements for the fine-grained behavior abstraction. A `ComponentInternalBehav-`

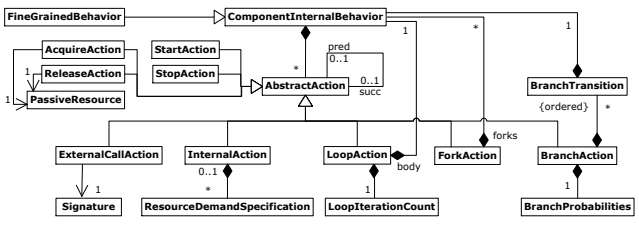

Figure 10: Fine-Grained Behavior Abstraction

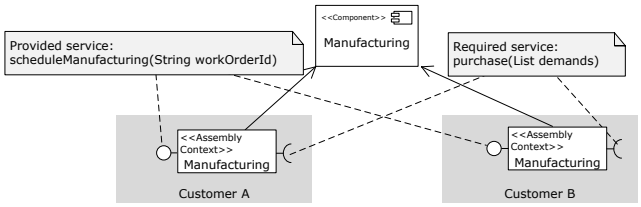

Figure 11: `Manufacturing` **Component**

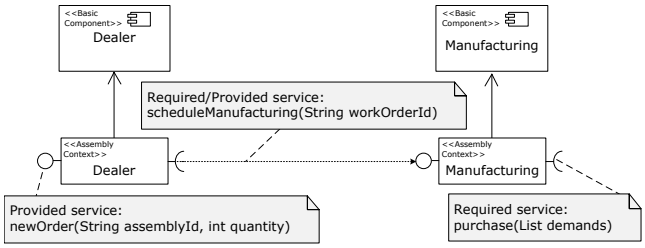

Figure 12: `Dealer` **and** `Manufacturing` **Components**

`ior` models the abstract control flow of a service implementation. Calls to required services are modeled using so-called `ExternalCallActions`, whereas internal computations within the component are modeled using `InternalActions`. Control flow actions like `LoopAction`, `BranchAction` or `ForkAction` are only used when they affect calls to required services (e.g., if a required service is called within a loop; otherwise, the whole loop would be captured as part of an `InternalAction`). `LoopActions` and `BranchActions` can be characterized with loop iteration counts and branch probabilities, respectively.

3.3 Modeling Parameter Dependencies

3.3.1 Motivation

Figure 11 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. Given that two different deployments of the application are involved, the respective probabilities for the two component instances of type `Manufacturing` can differ significantly. For instance, a customer with a large manufacturing site's warehouse will order parts less frequently than a customer who orders items "just in time".

As discussed in Section 2, 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 Figure 12 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 using `workOrderId` as a reference which is then passed to service `scheduleManufacturing` as an input parameter.

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

In such a case, provided that we know about the existence of the parameter dependency, we can use monitoring statistics collected at run-time to characterize the dependency probabilistically. Figure 13 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. Figure 13 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.

We conclude that 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.

3.3.2 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

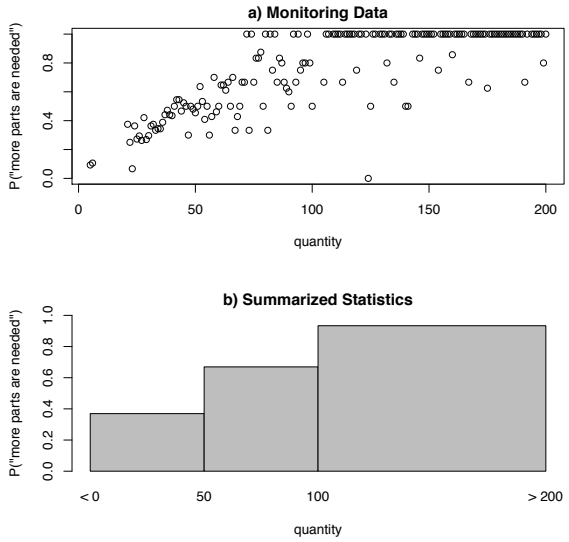

Figure 13: `scheduleManufacturing` **Statistics**

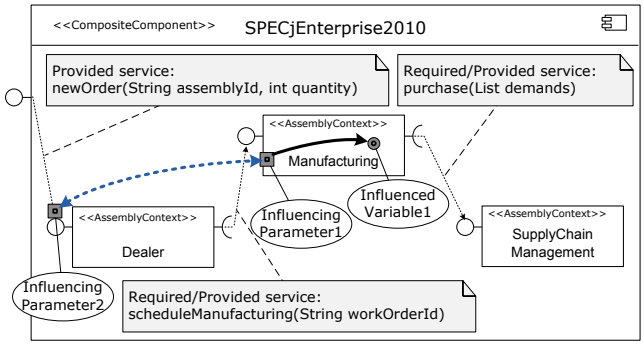

Figure 14: Modeling Parameter Dependencies

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.

Figure 14 illustrates our modeling approach in the context of the presented example from Figure 12. 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 `SPECjEnterprise-2010` 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 (Figure 13). Our modeling approach supports both empirical and explicit characterizations for both dependency and correlation relationships between model variables.

Note that an influencing parameter does not have to belong to a provided or required interface of the component. It can be considered as auxiliary model entity allowing to model parameter dependencies in a more flexible way. If an influencing parameter cannot be observed at run-time, the component's execution is obviously not affected, however, the parameter's influence cannot be taken into account in online performance predictions. The only thing that can be done in such a case is to monitor the influenced variable independently of any influencing factors and treat it as an invariant.

Finally, the same software component might be used multiple times in different settings, e.g., as in our scenario where the same application is run on behalf of different customers in separate virtual machines with customized application components. Hence, the meta-model should provide means to specify the scope of influencing parameters. A *scope* of an influencing parameter specifies a context where the influencing parameter is unique. This means, on the one hand, that measurements of the influencing parameter can be used interchangeably among component instances provided that these instances belong to the same context. On the other hand, it means that measurements of the influencing parameter are not transferable across scope boundaries. Thus, if monitoring data for a given influencing parameter is available, it should be clear based on its scope for which other instances of the component this data can be reused.

Information about parameter scopes is particularly important when using online performance models to predict the impact of dynamic reconfiguration scenarios. For instance, when considering the effect of adding server nodes to the application server cluster of a given customer (hosting instances of our `SPECjEnterprise2010` composite component), given that influencing parameters within the cluster belong to the same context, monitoring statistics from existing instances of the `SPECjEnterprise2010` component can be used to parameterize the newly deployed instances.

3.3.3 *Meta-Model*

Model Variables. The model variables involved in dependency specifications are divided into influenced variables and influencing parameters. As shown in Figure 15, 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 param-

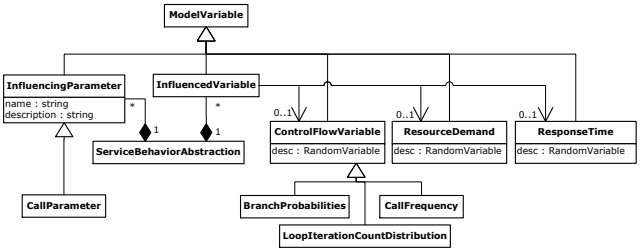

Figure 15: Model Variables

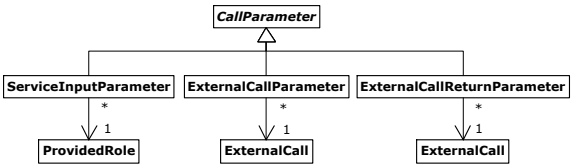

Figure 16: Call Parameter Hierarchy

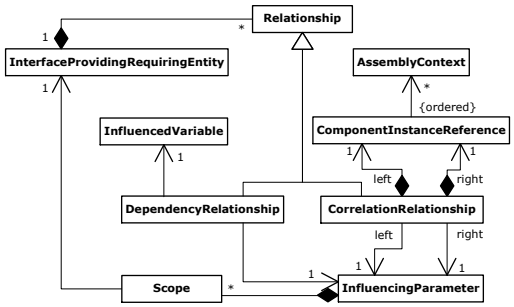

Figure 17: Relationships between `InfluencedVariables` and `InfluencingParameters`

eters or return parameters of external calls (see Figure 16) are special types of influencing parameters.

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

Relationships: Dependency and Correlation. As shown in Figure 17, 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.

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 Figure 14 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 Figure 17, 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) 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.

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 `EmpiricalChar-`

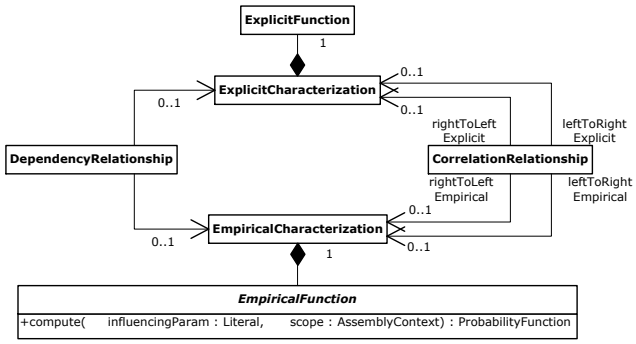

Figure 18: `ExplicitCharacterization` and `EmpiricalCharacterization`

`acterization` (see Figure 18). The former means that the relationship is characterized using explicit parameter dependencies as known from PCM. Such a characterization is suitable if an expression of the functional relation between the two considered model variables is available. If this is not the case, an `EmpiricalCharacterization` can be used to quantify the relationship using monitoring statistics. The entity `EmpiricalFunction` in the figure describes the interface to a characterization function based on empirical data. An `EmpiricalFunction` can for example be based on monitoring statistics as discussed earlier (Figure 13) and can be represented by abstracting the obtained statistics as illustrated in Figure 13 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 : `Literal` $\longrightarrow$ `ProbabilityFunction`. It maps a value of p to a probability function describing v. The signature of an empirical function $\hat{f}$ is $\hat{f}$: `Literal` $\times$ `AssemblyContext` $\longrightarrow$ `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.

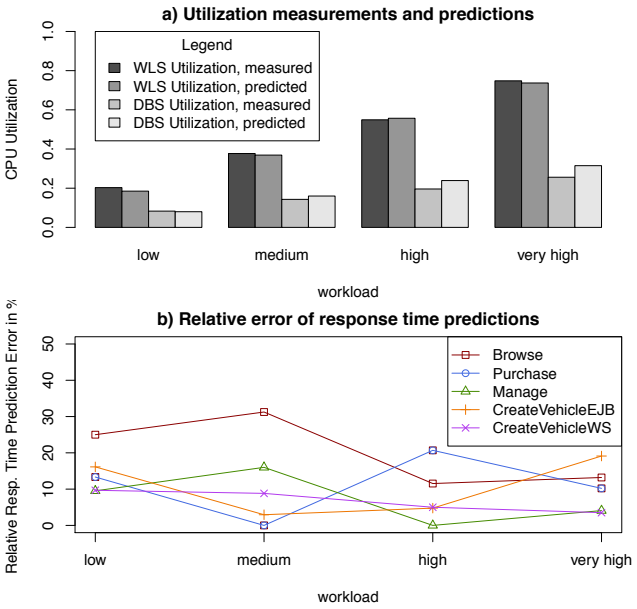

Figure 19: Exemplary Measurements and Prediction Results with SPECjEnterprise2010

4. SUMMARY OF EVALUATION

To evaluate the feasibility and effectiveness of our modeling approach, we deployed the benchmark in the system environment depicted in Figure 4 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 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 ($\approx$ 20% WLS CPU utilization), medium load conditions ($\approx$ 40%), high load conditions ($\approx$ 60%) and very high load conditions ($\approx$ 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 10ms to 70ms. We considered a number of different scenarios varying the application workload and system configuration. For lack of space, here we only show some exemplary results to illustrate the model accuracy. The measured and predicted server utilization for the different load levels are depicted in Figure 19 a). The utilization predictions fit the measurements very well, both for the WLS instances as well as for the DBS. Figure 19 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.

5. RELATED WORK

There have been presented several approaches on QoS and resource management based on online performance analysis [14, 10]. More recently, also approaches for the adaptation of virtualized systems using performance models have been proposed [10, 19]. Problem of all the approaches is that their performance analysis are rather restricted (if any) in terms of the level of detail. In general, the approaches do not consider run-time dynamics and aspects relevant to analyzing the performance behavior in an evolving environment or do not capture important parametric dependencies.

However, there exist many different design-time (component-based) performance modeling approaches that explicitly consider the influence of parameters in their performance analysis [1, 3, 23, 2]. The most advanced approaches concerning parametric dependencies are [1, 3]. 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 [2, 23] 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 parametrization. Also important to mention are transformation techniques which modify a given performance model by adding parametric connector behaviors [5] or parameterized performance annotations [8].

Orthogonal approaches tackling the challenge of parametric dependencies in performance analysis are [7, 20]. 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 main drawbacks of the presented approaches are that they are either limited to a subset of parameters, try to abstract from important details or even try to model the component's internal state completely instead of modeling parametric dependencies at a level of detail that is suitable for online performance analysis.

6. 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 approaches to modeling the component execution context are not suitable for use at run-time. They do not provide enough flexibility in the way parameter and context dependencies can be expressed and resolved. To address this issue, we proposed a novel performance meta-model for expressing and resolving parameter and context dependencies, specifically designed for use in *online* scenarios. 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.

As part of our on-going work, we are integrating the proposed meta-model with meta-models for modeling resource landscapes of distributed virtualized service infrastructures [9]. The presented modeling abstractions are an integral part of the Descartes Meta-Model (DMM) [11], a new meta-model that we are developing for run-time QoS and resource management in virtualized service infrastructures.

7. REFERENCES

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

[2] A. Bertolino and R. Mirandola. Cb-spe tool: Putting component-based performance engineering into practice. In *CBSE*, pages 233–248, 2004.

[3] E. Bondarev, J. Muskens, P. d. With, M. Chaudron, and J. Lukkien. Predicting real-time properties of component assemblies: A scenario-simulation approach. In *EUROMICRO*, pages 40–47, 2004.

[4] F. Brosig, N. Huber, and S. Kounev. Automated Extraction of Architecture-Level Performance Models of Distributed Component-Based Systems. In *26th Int. Conf. on Automated Software Engineering*, 2011.

[5] V. Grassi, R. Mirandola, and A. Sabetta. A model transformation approach for the early performance and reliability analysis of component-based systems. In *CBSE'06*.

[6] V. Grassi, R. Mirandola, and 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(4), 2007.

[7] D. Hamlet. Tools and experiments supporting a testing-based theory of component composition. *ACM Trans. Softw. Eng. Methodol.*, 18:12:1–12:41, 2009.

[8] J. Happe, H. Friedrichs, S. Becker, and R. Reussner. A configurable performance completion for message-oriented middleware. In *WOSP*, 2008.

[9] N. Huber, F. Brosig, and S. Kounev. Modeling Dynamic Virtualized Resource Landscapes. In *QoSA 2012*.

[10] G. Jung, M. Hiltunen, K. Joshi, R. Schlichting, and C. Pu. Mistral: Dynamically Managing Power, Performance, and Adaptation Cost in Cloud Infrastructures. In *ICDCS*, 2010.

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

[12] S. Kounev, F. Brosig, N. Huber, and R. Reussner. Towards self-aware performance and resource management in modern service-oriented systems. In *SCC 2010*.

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

[14] J. Li, J. Chinneck, M. Woodside, M. Litoiu, and G. Iszlai. Performance model driven QoS guarantees and optimization in clouds. In *CLOUD '09*, Washington, DC, USA, 2009. IEEE.

[15] D. A. Menascé and H. Gomaa. A method for design and performance modeling of client/server systems. *IEEE Trans. Softw. Eng.*, 26(11), 2000.

[16] R. Nou, S. Kounev, F. Julia, and J. Torres. Autonomic QoS control in enterprise Grid environments using online simulation. *Journal of Systems and Software*, 82(3), 2009.

[17] Object Man. Group (OMG). UML Profile for Modeling and Analysis of Real-Time and Embedded systems (MARTE), 2006.

[18] G. Pacifici, W. Segmuller, M. Spreitzer, and A. N. Tantawi. Cpu demand for web serving: Measurement analysis and dynamic estimation. *Performance Evaluation*, 2008.

[19] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated control of multiple virtualized resources. In *Proc. of EuroSys '09*. ACM, 2009.

[20] M. Sitaraman, G. Kulczycki, J. Krone, W. F. Ogden, and A. L. N. Reddy. Performance specification of software components. In *SSR'01*, 2001.

[21] C. U. Smith, C. M. Llado, V. Cortellessa, A. Di Marco, and L. G. Williams. From UML Models to Software Performance Results: an SPE Process based on XML Interchange Formats. In *WOSP*, 2005.

[22] SPECjEnterprise2010 Design Document. `http://www.spec.org/jEnterprise2010/docs/DesignDocumentation.html`. 2011.

[23] X. Wu and M. Woodside. Performance modeling from software components. In *WOSP*, pages 290–301, 2004.