Online Model Learning for Self-Aware Computing Infrastructures

Simon Spinner, Johannes Grohmann, Simon Eismann, Samuel Kounev

University of Würzburg
Am Hubland, 97074 Würzburg, Germany
Email:[first_name].[last_name]@uni-wuerzburg.de

Abstract

Performance models are valuable and powerful tools for performance prediction. However, the creation of performance models usually requires significant manual effort. Furthermore, as the modeled structures are subject to frequent change in modern infrastructures, such performance models need to be adapted as well. We therefore propose a reference architecture for online model learning in virtualized environments, which enables the automatic extraction of the aforementioned performance models. We follow an agent-based approach, which enables us to incorporate the extraction of information about the application structure as well as the virtualization structures present in modern computing centers. Our evaluation shows that our collaborating agents are able to reduce the manual effort of performance model extraction by 85.4%. The resulting performance model is able to predict the system utilization with an absolute error of less than 4% and the end-to-end response time with a relative error of less than 21%.

Keywords: Self-Aware Computing, Performance Model, Model Extraction, Model Learning

1. Introduction

IT services hosted in data centers, such as public internet services (e.g., Netflix, Facebook, or Google) as well as intranet services in corporate networks, are typically subject to time-varying workloads.

Changes in the current number of users or their interactions with a service influence its resource demand. At any point in time, the amount of resources allocated to an application needs to fulfill its current demand. As a result, continuous adaptations to the resource allocations of an application are required during operation.

Current approaches to automatic resource management in industry are based on a rule-based approach: A system administrator manually defines custom triggers that fire when a metric reaches a certain threshold (e.g., high resource

utilization or load imbalance) and execute certain reconfiguration actions. However, application-level performance metrics, such as response time, normally exhibit a highly non-linear behavior on system load. Therefore, it is not possible to determine general thresholds of when triggers should be fired, given that the appropriate triggering points are typically highly dependent on the architecture of the hosted services and their usage profiles, which can change frequently during operation.

In order to overcome the limitations of rule-based approaches, the usage of different kinds of performance models has been proposed in the literature [1, 2]. Stochastic performance models (e.g., queueing networks, or descriptive modeling languages such as DML [3]) provide powerful abstractions of a combined hardware and software system describing its performance-relevant structure and behavior. They can predict the impact of a reconfiguration action on the system performance in advance and thus promise significant improvements for the automatic resource management of IT services. Existing model-based approaches either abstract the application as a black-box severely limiting their prediction capabilities; or expect manually created model instances as input. However, the expertise and effort required to create and maintain such detailed models of the infrastructure and applications in a virtualized environment manually pose a major challenge to exploit the advanced prediction capabilities of stochastic performance models for automatic resource management.

Challenges. In this article, we describe a new agent-based reference architecture enabling the deep integration of online model learning capabilities into virtualized environments. Our reference architecture addresses the following challenges:

- Given that model learning is performed during system operation, the system workload and configuration cannot be controlled. We rely on empirical observations while applications are serving production workloads. In order to avoid significant overheads on the performance of services, existing monitoring infrastructures and platform interfaces should be used to obtain the empirical information required for model learning.

- The integration of model learning capabilities into systems requires a profound understanding of the system architecture – including the application and any platform layers – and at the same time a deep knowledge of performance modeling techniques. However, system administrators often do not have sufficient skills to perform such tasks. Furthermore, it can be time-consuming and costly to design and implement model learning capabilities for a given system. Therefore, ways to enable the reuse and sharing of model learning capabilities between systems are necessary.

- Multiple applications with diverse technology stacks typically share the same underlying infrastructure in virtualized environments influencing each other. A performance model needs to represent the complete virtualized system (including the different applications) integrating information from heterogeneous datasources. However, the deployment of

applications and their software stacks are often not known before system run-time (especially with the advancement of on-demand provisioning of Virtual Machines (VMs) in cloud environments). As a result, the end-to-end performance model of the system can only be dynamically composed a system run-time.

- The deployment and configuration of applications may change frequently due to automatic or manual reconfigurations (e.g., deployment of new VMs, or migration of existing ones). As a result, the overall performance model of the system needs to be continuously updated to always reflect the current system architecture and configuration.

A major field of research is the automatic extraction of performance models based on static and dynamic analysis of the system implementation and configuration in order to ease the usage of performance models. Existing work either describes holistic approaches to extract complete performance models, but assume a very specific technology stack [4, 5], or focuses on improving certain aspects of it (e.g., resource demand estimation [6]).

Contributions. In this paper, we propose an agent-based reference architecture for online model learning in virtualized environments. In particular, this paper makes the following contributions:

- We extend the notion of Virtual Appliances (VAs) to include model learning capabilities in order to automatically build and maintain submodels describing performance behavior of the application architecture and infrastructure layers.
- We introduce additional components into the virtualization platform to collect these sub-models and dynamically compose them into an end-to-end performance model of the complete system.
- We identify different roles an agent may take over during model learning and describe the required communication between agents in different roles.
- We develop an algorithm for merging the different model skeletons into a complete performance model in a central repository.

In order to evaluate this, we create a reference implementation of the proposed agent structure monitoring a distributed SPECjEnterprise2010 benchmark. We evaluate the degree of automation for the model extraction as well as the prediction accuracy of the resulting model. Although targeted at virtualized environments, some aspects, like the use of agent-based architectures for performance model extraction of distributed software systems and the proposed model merging algorithm, can be transferred to other application domains.

Structure. The remainder of the article is organized as follows. Section 2 introduces our proposed reference architecture for integrating model learning capabilities into virtualization platforms. Section 3 gives an overview of the related work. Section 4 describes our reference implementations of three different model learning agents. We evaluate our reference implementation in Section 5. Lastly, we summarize and conclude our work in Section 6.

2. A Reference Architecture for Online Model Learning

Modern hypervisors (e.g., VMware ESX or Xen) and virtualization management software (e.g., VMware vCenter) - in the following, the combination of both is called the *virtualization platform* - rely on standardized formats for VM images to support the deployment of new VMs. However, this image format is focused on the specification of the virtual hardware resources including their configuration and lacks meta-data describing the platform and application layers inside a VM. The program code of the platform and application layers, as well as any additional data, is contained in an unstructured binary image of the virtual hard disk.

Therefore, a virtualization platform is generally not aware of what is contained inside a VM. Although, the virtualization platform may access all data in the main memory and hard disks of a VM, the data is hard to interpret given that no general assumptions can be made on their structure. An approach to model learning solely based on information available in the virtualization platform inevitably leads to performance models abstracting application and platform layers as a black-box. In contrast, an approach based on model learning inside a VM may provide detailed performance models of the platform and application layers running in the same VM. However, in the latter case access to the underlying infrastructure layers or co-located applications is prohibited. In the following, we describe our reference architecture for model learning that bridges this gap.

2.1. Conceptual Overview

We argue that model learning capabilities should be integrated deeply into both the virtualization platform and the hosted VMs enabling the extraction of end-to-end performance models covering the virtual infrastructure, as well as any platform and application layers within VMs. We assume a virtualization platform that hosts a set of *VAs*. A VA is a set of pre-packaged VM images each containing a complete software stack ready to run on a virtualization platform VAs can significantly reduce the effort and knowledge required for deploying software systems. VAss are either provided directly by software companies or by individuals.VAs are built by experts of the respective system and can then be shared with others (e.g., through online marketplaces, such as VMware Solution Exchange[1]). When deploying such a VA, only certain pre-defined settings

[1] `https://solutionexchange.vmware.com/store`

may need to be customized (e.g., through a web interface provided by the VA) in order to adapt it to a target virtual environment (e.g., IP address settings, or passwords).

Our reference architecture is based on an extension of conventional VAs to include additional logic for learning performance models of the application as well as any contained platform layers (e.g., middleware systems or Java VMs) during system run-time. The model learning logic is encapsulated in specialized *agents* distributed as part of a VA. On instantiation of such a VA in a virtualized environment, the contained agent will start to monitor the application serving real production workloads and will automatically build a sub-model (so-called *model skeleton*) describing the observed performance behavior of the application and platform layers inside the VA. The agent continuously updates the model skeleton to reflect dynamic changes, for instance, in the configuration or the workload of an application. A virtualization platform may access the model skeletons extracted by the agents of a VA using a defined interface in order to obtain fine-grained performance models of an application.

Model skeletons created within a VA do not contain information about the underlying infrastructure layers, or co-located VAs as such information is not visible inside guest VMs. Therefore, the virtualization platform itself needs to contain agents that extract model skeletons of the data center and the virtualization platform. The virtualization platform then composes the model skeletons from different VAs and underlying infrastructure layers into an end-to-end performance model.

A VA may contain a complete application (e.g., a SAP ERP system, or a Zimbra Collaboration server), or provide only certain platform layers (e.g., a Java Enterprise Edition (Java EE) application server) on which custom applications can be deployed after VA instantiation. In the former case, the deployment of the VA typically involves only certain customizations of the configuration and the creator of the VA may be able to determine large parts of the model skeleton in advance. In the latter case, no prior knowledge about the static structure and dynamic behavior of applications running on the platform layers can be assumed and the extraction logic needs to create the model skeleton dynamically by analyzing the executed application.

Our reference architecture allows for agents to be specifically designed for a given software stack in a VA. Thus, it is possible to incorporate technology-specific prior knowledge into the model learning logic. The model skeletons may be partially or completely created at run-time based on dynamic system information obtained through sensor or reflection interfaces. Sensors provide empirical observations of the dynamic behavior of a system. Reflection describes the ability of a software system to determine its own structure and state (e.g., based on configuration files, byte-code, etc.). Both sensors and mechanisms for reflection on the application level are typically very technology-specific, and therefore are part of the VA.

System administrators, who deploy a VA in a virtualized data center, do not need to be experts in performance modeling. The model learning runs transparently in the background without disturbing the system operation. The resulting

end-to-end performance model of the virtualized system can be used for online resource management in conjunction with advanced reasoning techniques exploiting knowledge of the system architecture [7].

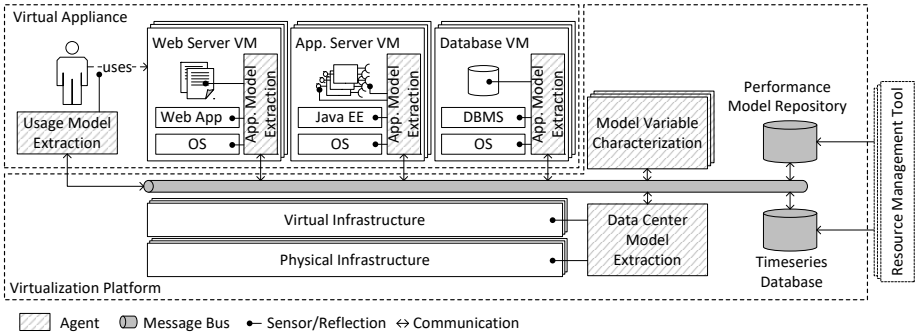

Figure 1: Conceptual overview of the reference architecture.

Components. Figure 1 gives an overview of the main components of our reference architecture. Our reference architecture relies on specialized agents focusing on learning models of certain aspects of a system:

- Each VA contains one or several *application model extraction* agents creating models of the application components including their behavior, their assembly, and their deployment, as well as any platform layers running inside a VM. These agents only determine the model structure and do not determine concrete values of model variables (e.g., resource demands).

- The *usage model extraction* agent focuses on the behavior of external users. It determines usage behaviors for different types of users and characterizes their load intensity. The agent needs to be able to observe incoming requests at the interface roles accessible from outside an application.

- The agents for *data center model* extraction are part of the virtualization platform so that they have access to the virtual and physical infrastructure layers in a data center. The agents are specifically designed for the infrastructure technologies in a data center, and must not make any assumptions on the software stack within VAs.

- Agents for *model variable characterization* implement generic dynamic analysis techniques to determine the current value of model variables based on empirical data. The empirical data may be provided by the VAs (e.g., throughput or response time measurements) or the virtualization platform (e.g., resource utilization statistics). Model variable characterization agents may not make any assumptions on the software stack running in a VA. These agents need to derive the information they require from the model skeletons and monitoring data provided by the model extraction agents in the VAs and the virtualization platform.

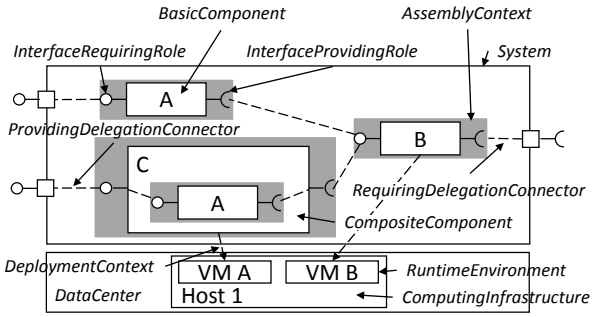

Figure 2: DML overview.

For communication purposes, we assume that all agents have access to a shared network. The network connects them with the central components provided by the virtualization platform. A *message bus* connects all agents and decouples them using asynchronous publish/subscribe communication facilities. The *performance model repository* merges the model skeletons coming from different agents in VAs and the virtualization platform into a consistent end-to-end model. Resource management tools may access the current model version in the repository for reasoning purposes. The resource management tools are not part of our reference architecture. If historic monitoring data needs to be persisted for longer periods of time, *time series databases* may be used to collect and store such data.

2.2. Meta-model

The model skeletons and the end-to-end performance model share a common *meta-model* providing a formal definition of an abstract syntax. The meta-model enables us to create technology-independent descriptions of the system architecture. Furthermore, a common meta-model helps to enforce consistent syntactic and semantic constraints between model skeletons and simplifies their composition into an end-to-end performance model avoiding the need for model transformations.

Our reference architecture is based on an existing meta-model for online resource management, called Descartes Modeling Language (DML) [7]. We will give a short introduction here.

The Descartes Modeling Language (DML) is a descriptive, architecture-level performance model specifically targeted at online performance and resource management in data centers and offers flexible solution techniques based on model-to-model transformations, e.g., to Queueing Networks (QNs) or Queueing Petri Networks (QPNs). Furthermore, in contrast to other architecture-level performance models, it supports empirical as well as explicit descriptions of model variables and parameter dependencies. For a complete specification of DML see [8, 9].

A DML instance (see Figure 2) contains a *repository* of *basic* and *composite components*. Each component has *interface providing* and *interface requiring roles*. Roles are associated with an *interface* that declares a set of *operations*. Each operation of an interface providing role corresponds to a service of a component that can be called by other components. The interface requiring roles specify the services that a component depends on. A basic component must specify a service behavior for each provided service (i.e., for each interface providing role and operation). The service behavior specifies the performance relevant control flow of the component (i.e., resources accesses, external calls to other services, loops, forks, etc.). Composite components bundle a set of components which are deployed together.

Components are composed to a *system* using *assembly contexts*, *assembly connectors*, and *delegation connectors*. Each assembly context represents a component instance within a system or a composite component. A component may be instantiated multiple times in a system at different positions in the control flow (e.g., component `A` in Figure 2). Assembly connectors represent the control flow between components. Delegation connectors can be used to expose providing or requiring roles to enclosing composite component or system.

The *resource landscape* describes the physical and logical resources in a *data center*. The main entity are *containers* which can be a *computing infrastructure* (i.e., physical server) or a *runtime environment* (e.g. a VM or a middleware service). Each container contains a description of its resources (CPU, hard disks, network links, etc.). *Deployment contexts* map an assembly context to a container.

A *usage profile* contains a set of *usage scenarios* describing the incoming workload to a system (open/close workload). A usage scenario defines the sequence of *system user calls* to interfaces provided by the system.

Compared to low-level prediction models (e.g., QNs), a descriptive metamodel provides the advantage of greater expressiveness to include additional information on the static architecture and dynamic behavior of a system. Compared to other descriptive architecture-level performance models, such as Palladio Component Model (PCM) [10] and SLAstic [11], Descartes Modeling Language (DML) provides a number of benefits:

- DML provides explicit modeling elements to describe the layering and configuration of the system environment. This is important to capture the structure of the underlying virtualized infrastructure.
- DML supports different levels of granularity to describe the behavior of services (i.e., black-box, coarse-grained, and fine-grained). The different granularity levels increase the flexibility when solving a model to tradeoff between prediction time and accuracy (see [3]).
- Model variables (e.g., resource demands or branching probabilities) and parameter dependencies can be marked as explicit or empirical in DML. Explicit model variables are assumed to have a fixed value (or a stochastic expression calculating a value based on input parameters). The values

of empirical model variables are determined at run-time based on monitoring data from the system. We exploit this modeling construct for our model skeletons.

2.3. Model Extraction Agents

Model extraction agents may take on different roles in our reference architecture. An agent role defines the aspects of the performance model an agent takes care of. In order to extract a complete model, all required roles need to be implemented by agents. We analyzed the DML meta-model identifying clusters of classes in the meta-model with a high cohesion. Such clusters should be covered by a single agent role in order to reduce the required communication between agent roles. In the following, we describe each agent role and its interfaces to other roles. The agent roles are grouped by extraction scopes.

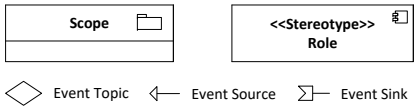

Figure 3: Notation.

Notation. The communication between agents in our reference architecture is based on asynchronous, event-based communication. In the following, we adopt the Unified Modeling Language (UML) component diagram notation with the profile for event-based communication used by Rathfelder [12]. Figure 3 gives an overview of the notation. Each agent role is represented by a component. Composite components are used to group them into extraction scopes. A component may have any number of event sources and event sinks specifying the types of events it may send and receive. Event topics connect event sources with event sinks and allow for publish/subscribe communication. Events are either change notifications or scope delegations, if the event source is marked with the stereotype `<<delegates>>`.

2.3.1. Data Center Scope

Agents in the data center scope create and maintain a resource landscape model of the physical hardware infrastructure as well as a high-level system model of the applications running in a data center. The agents in this scope treat the physical hardware nodes and applications as black-boxes without any knowledge of their internal structure. Figure 4 gives an overview of the agent roles in the data center scope:

D1 The agent discovers the global, static structure of the data center. It identifies compute and storage nodes representing physical computers and storage systems in a data center.

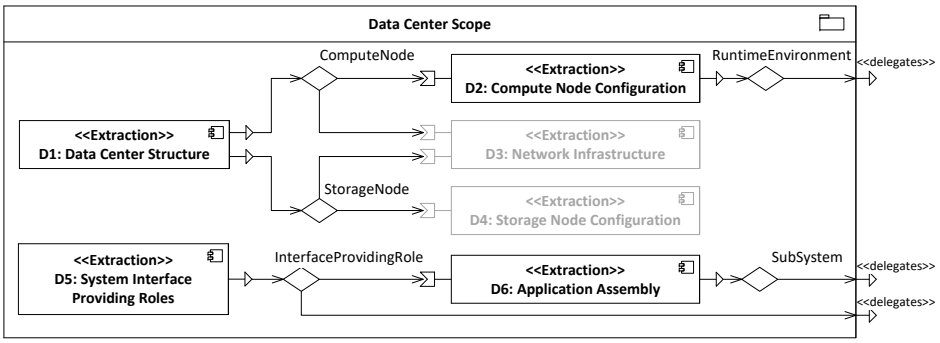

Figure 4: Overview of the data center scope.

D2 The agent enriches compute nodes identified by *D1* with information on their configuration (e.g., number of CPUs and their speed). Furthermore, it determines the directly contained runtime environment (e.g., a hypervisor, or a native operating system).

D3/D4 The agent extracts the configuration of storage nodes identified by *D1* and the network infrastructure respectively. The current version of DML does not provide meta-models to describe these aspects in detail. Future work may integrate the preliminary works of Noorshams [13] and Rygielski [14]. These roles are optional.

D5 An agent with this role is responsible to identify services provided by applications to users outside the data center including individual operations and their input and output parameters.

D6 The agent determines the different applications running inside a data center including their interface providing and requiring roles. The interface providing roles of an application describe the services which are publicly visible to other applications in the same data center or to external users. The interface requiring roles of an application may be connected to services provided by other applications in the same data center.

2.3.2. Usage Scope

The usage model captures the external requests of an application coming from outside the data center. Load from other applications in the same data center is covered by agent role *D6*. Compared to the workload characterization step in classic performance modeling, we only cover the arrival process in the usage model and do not consider the mapping to resources as part.

In order to extract usage models agents require empirical information on the type and frequency of requests. In case of session-based workloads, we assume that agents have access to session logs containing the information required to correlate individual requests. Figure 5 gives an overview of the agent roles in the usage scope:

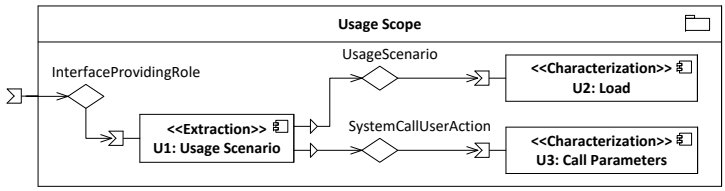

Figure 5: Overview of the usage scope.

U1 Users of a system may differ in their usage behavior (i.e., number and types of system calls) and the load they cause on the system (i.e., load intensity and open vs. closed workloads). Agents group user sessions into usage scenarios with similar characteristics and determine a probabilistic model of the usage behavior (i.e., the sequence of requests) for each scenario.

U2 The agent needs to characterize the workload type (open vs. closed) as well as the load intensity over time, including models describing load fluctuations, such as seasonal patterns, trends, and bursts.

U3 The performance behavior of an application may depend on the value of input parameters. In order to extract such parameter dependencies, agents may collect the values of parameters of individual requests to enabling the characterization of parameter dependencies (see roles *A6* and *A7*). This role is only needed if parameter dependencies are considered.

2.3.3. Platform Scope

On top of the physical hardware layer (represented by a data center scope), data centers typically have one or several platform layers representing the hypervisor and optional middleware layers required by applications. Each platform layer may host agents that extract models describing its structure and behavior. Figure 6 gives an overview of the agent roles in a platform scope:

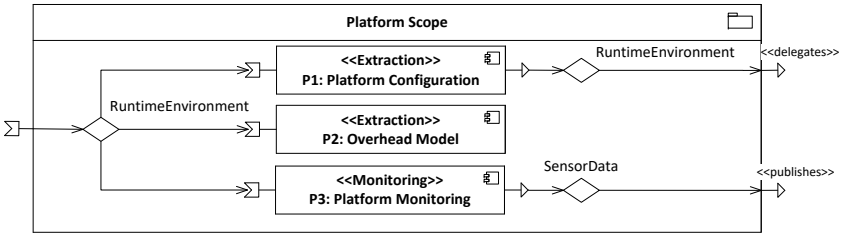

Figure 6: Overview of the platform scope.

P1 The agent analyzes a run-time environment (e.g., a hypervisor, or a middleware system) and determines its logical software resources (e.g, vCPUs, thread and connection pools, or asynchronous message queues), as well

as any contained run-time environments (e.g., VMs). The agent extracts the current configuration of software resources of the run-time environment (e.g., the resource capacity, or scheduling priorities).

P2 The agent determines the dynamic behavior of a run-time environment, i.e., its performance impact on higher layers in a system. For instance, hypervisors introduce certain overheads slowing down the processing within a VM [15]. Furthermore, resource-intensive reconfigurations may impact the performance of VMs in a data-center. These types of overheads may be optionally captured with explicit models.

P3 At system run-time, agents expose monitoring statics provided by platform layers covering the state of physical and logical resources (e.g., the current resource usage) in a technology-independent way. This is an optional role.

2.3.4. Application Scope

The application model extraction covers the static structure and dynamic behavior of an application. We assume a component-based software architecture. The agents in these roles focus on determining all possible control flow paths in a component and do not determine the value of control flow variables (e.g., branching probabilities, or external call frequencies) and resource demands. These model variables are characterized by agents in the model variable scope. Figure 7 gives an overview of the agent roles in the application scope:

A1 The agent discovers the software components an application consists of. This includes the interface providing and requiring roles of the components as well as interface definitions (i.e., signatures and parameters). The components are extracted on a type level, i.e., only one component definition is created even if it is used in different assembly contexts.

A2 Agents determine the composition of components identified by A1 within an application. They discover all component instances and determine the control between them.

A3 The agent determines the deployment of component instances identified by *A2* on runtime environments identified by *P1*.

A4/A5 Role *A1* only covers a black-box service behavior description. If more fine-grained instrumentation is supported, agents can optionally create more detailed service behavior descriptions. A coarse-grained one covers the resources that are accessed and components called by a component irrespective of the order. A fine-grained description consists of individual actions the component-internal control flow consists of (e.g., internal actions, forks, loops, and branches) including their exact execution order.

A6/A7 Model variables (e.g., resource demands or branching probabilities) may depend on the value of an input parameter. Agents identify possible parameter dependencies within a single component (e.g., which input parameters have an influence on the resource demand of a component) and

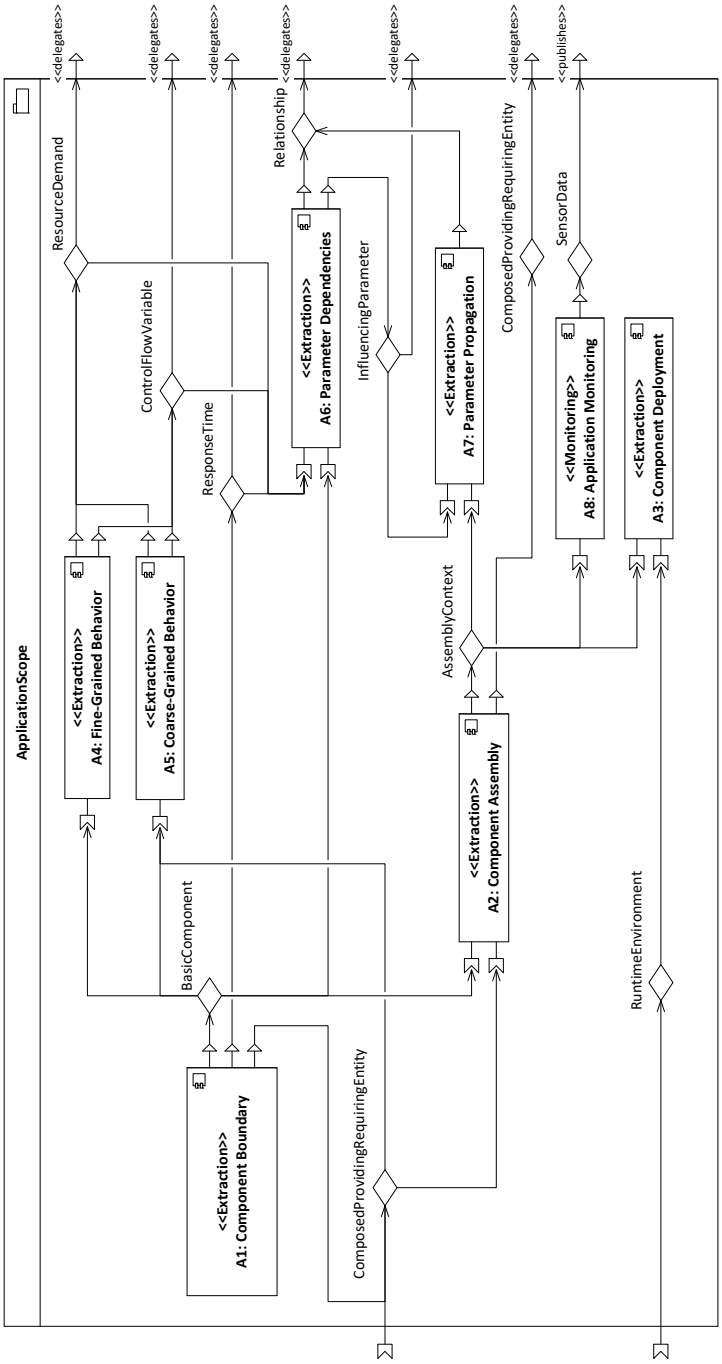

Figure 7: Overview of the application scope.

determine the data flow of input parameters between components (*A7*). This is an optional role.

A8 The agent collects application-level performance statistics (e.g., response times and throughput of services) using instrumentation techniques provided by the application.

2.3.5. Model Variable Scopes

The techniques to characterize model variables are typically generic, however, they may require access to information in the data center scope and the platform scopes. Therefore, agents for the characterization of model variables run in scopes separate to the application scope. Such agents may also use statistical techniques based on empirical monitoring with higher computational complexity and may be deployed on isolated machines.

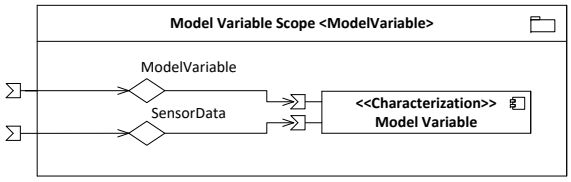

Figure 8: Overview of the model variable scope.

Figure 8 gives an overview of the model variable scope. The scope definition is a generic template where the `ModelVariable` parameter determines the actual variable type: `ResponseTime`, `ControlFlowVariable`, `ResourceDemand`, `InfluencingParameter`, or `Relationship`. We distinguish between five agent roles accordingly:

M1 Black-box service behaviors contain a function describing the response time of a component service depending on input parameters. The agent derives a function describing the response time depending on the values of input parameters.

M2 Resource demands are required for coarse-grained and fine-grained service behaviors. The agent determines a value for resource demands (including their stochastic distribution).

M3 The agent determines values for control flow variables, such as loop iteration counts or branching probabilities in fine-grained service behaviors and external call frequencies in coarse-grained ones. The value of control flow variables may depend on values of input parameters.

M4 In order to enable the characterization of parameter dependencies, the distribution of values of input parameters to a component service needs to be determined.

M5 The agent determines the data flow of input parameter values between components in an application and provides empirical distributions of these relationships.

The model variable scopes are only required if the agent roles in the application scope that identify the corresponding model variables are present.

2.4. Performance Model Repository

The model repository is the central place where the extracted DML model instance of the complete system is maintained and persisted. The model instance is the result of merging all model skeletons coming from different agents into a single model instance. Model skeletons from different agents may overlap, i.e., they may contain model objects referring to the same physical entity. To avoid duplication of model objects, a merging of the model skeletons is required resolving duplicate model elements to a single element in the performance model repository.

2.4.1. Model Skeleton Composition

Formal Definitions. For the following descriptions, we adopt the formalization of Eclipse Modeling Framework (EMF) models used in [16]. We give a short repetition of this formalization here.

Definition 1 (Meta-model)**.** A meta-model "is a tuple $em = (C, D, F, P)$. The components of which are sets of classes, data types, features and properties, respectively. F is partitioned into sets of attributes A and references R ($F = A \cup R, A \cap R = \emptyset$)"[16]. P consists of attribute functions describing the properties of classes, data types and features (such as $superTypes$, $domain$, $range$, $many$, $ordered$, $unique$, $containment$, and $opposite$).

A model consists of a set of objects which are instantiations of classes in a meta-model. Each object contains values for the features defined in its class. Attributes have literal values defined by their data type. References link to other objects in the model.

Definition 2 (Model)**.** "Let $em = (C, D, F, P)$ denote a meta-model. A model instantiated from em is a tuple $m = (O, class, FV)$. O is a set of objects. $class : O \rightarrow C$ assigns to each object the class from which it was instantiated. For each feature $f \in F$ there is one and only one feature value function $fv \in FV$."[16]

Algorithm Overview. Suppose a model s sent by an agent as part of a model skeleton and the global model p in our model repository, the goal is to merge the contents of s into the existing version of p. We use a state-based, two-way merging algorithm. State-based means that it does not require a complete change history; the merging is done solely based on the latest versions of s and p. We start with an empty model p. Each new or updated model s sent by an agent to the model repository is then merged into the current version of p in an atomic transaction. The merge algorithm needs to address the following two steps:

- *Differencing:* Find the model objects that are contained in both models. Given that the model skeletons are created independently by different agents, we need to consider strategies for *matching* the same objects in s and p.

- *Merging:* If a model object is only contained in s it can be simply copied to p. Otherwise, we need to merge the two objects to integrate any changes of s into p.

Compared to other merging algorithms [e.g. 16] that assume the two model versions come from a common base version, our algorithm differs as the model skeleton s only represents a subset of p. As a result, it is not possible to reliably determine model elements deleted from s solely on the current version of s and p. Model objects that are in p but not in s cannot be deleted directly because other agents may still reference it in their model skeletons. Therefore, the model repository implements a reference-counting scheme. For each model object in p, the model repository maintains the set of agents that referenced it in their latest version of the model skeleton.

Object Matching. DML defines the abstract base class `Identifier` with a string attribute `id`. The `id` is used to uniquely identify a model object on a global level in a DML model instance. However, the `id` attribute is not suitable for matching model objects. Model objects referring to the same physical entity may have different values for the `id` attribute if created by different model extraction agents. The `id` attribute is typically randomly generated. Furthermore, not all classes in DML are subclasses of `Identifier`.

Instead of the `id` attribute, we use matching rules specific to the DML meta-model. These rules specify a set of identifying features (i.e., attributes or references) for each class in the DML meta-model. The values of these features need to be unique only on a local level (i.e, between siblings in the containment tree). The values of the `id` attribute in a model skeleton are always ignored when merging them into the model repository.

Conflict Prevention. Conflicts may occur if two different model skeletons contain the same elements. Two models em' and em'' are conflicting, if they contain a model object $o \in (O' \cap O'')$ with a feature f for which the feature value functions fv'_f and fv''_f assign different values. The feature may be an attribute, a containment or a cross-reference. The equality of feature values depends on the feature attributes. For single-value features, the two values are compared directly. For unordered many-value features, we check for set equality. For ordered multi-value features, we also compare the sequence of values.

Given two versions of a model, which were changed independently of each other, there exists no domain-independent merging algorithm for models that can resolve all types of conflicts automatically [17]. Therefore, we need to introduce additional constraints to prevent conflicts. The basic idea is to allow sharing of model objects between different model skeletons only if we can be sure that any conflicting changes to these objects can be resolved automatically.

To enforce this constraint, we rely on the agent roles introduced in Section 2.3. In Section 2.5, we describe the conditions that need to be fulfilled to allow automatic merging of model objects.

2.5. Merge Algorithm

We now derive a formal description of the state of the model repository and the model skeletons. These descriptions are focused on model merging and are not complete formalizations. They assume the availability of a common meta-model $em = (C, D, F, P)$, which is DML in our case.

Definition 3 (Model Skeleton)**.** A model skeleton $m_s = (s, b, id, owns)$ contains a model $s = (O_s, class, FV)$ conforming to the meta-model em. The element $b \in B$ specifies the agent that created the model skeleton. In addition, it provides a function $id : C \rightarrow \text{SET}(F)$ that returns a set of identity features used for matching elements. The function $owns : O_s \rightarrow boolean$ specifies whether an object is owned ($true$) or only referenced ($false$) in the model skeleton.

Several model skeletons are merged into a central model repository. The state of a model repository is defined as:

Definition 4 (Model Repository)**.** The state of a model repository is defined by a tuple $m_p = (p, B, shared, refs, owner)$ containing a model instance $p = (O_p, class, FV)$ conforming to the meta-model em. The set B contains all currently connected agents. The function $shared : O_p \rightarrow boolean$ specifies whether multiple owners are allowed for an object. The function $refs : O_p \rightarrow SET(B)$ returns a set of agents that reference a model object. The function $owners : O_p \rightarrow B$ determines the set of agents that own a model object.

The merging is based on a two-way merging algorithm: the model versions are s and p. It is important to note that compared to traditional merging algorithms, s is only a subset of p.

Differencing. The first step, is the differencing to determine the set of changes in the model skeleton m_s that need to be merged into the model repository m_p. We define a helper function $matches : O_s \times O_p \rightarrow boolean$. The function evaluates to true if the following condition for two objects $o_1 \in O_s$ and $o_2 \in O_p$ with $c = class_s(o_1) = class_p(o_2)$ is fulfilled: $\forall i \in id(c) : fv_i(o_1) = fv_i(o_2)$. The results of the differencing step are the following three sets (b is the agent that created model skeleton m_s):

- The new objects set contains all objects in the model skeleton which have no matching counterparts in the repository.

$$\Delta_{new} = \{o_s \in O_s \mid \forall o_p \in O_p : \neg matches(o_s, o_p)\}$$

- The existing objects set contains all objects newly added to a model skeleton which already have matching counterparts in the repository, e.g. created by another agent.

$$\Delta_{exists} = \{o_s \in O_s \mid \exists o_p \in O_p : (matches(o_s, o_p) \wedge b \notin refs(o_p))\}$$

- The removed objects set contains all objects which were contained in a previous version of a model skeleton and now have been removed by the agent. We perform reference counting to ensure that no objects are deleted in the model repository which are still referenced in any of the model skeletons.

$$\Delta_{remove} = \{o_p \in O_p \mid b \in refs(o_p) \wedge (\forall o_s \in O_s : \neg matches(o_s, o_p))\}$$

Merging. The merging step uses the sets Δ_{new}, Δ_{exists}, and Δ_{remove} from the differencing step and merges the model skeleton m_s into the model repository m_p. The result is a new version m'_p of the model repository. The merging uses the following four primitives:

- $create(o_s)$: Creates a new object o_p in the model repository m'_p that matches the object o_s in the input model skeleton. The post-condition of this function is:

$$\exists o_p \in O'_p : (matches(o_s, o_p) \wedge b \in refs'(o_p) \\ \wedge (owns(o_s) \implies b \in owners'(o_p)))$$

b is the agent which created the model skeleton. We update the $refs$ and $owner$ attribute functions of the model repository accordingly.

- $link(o_s, o_p)$: Similar to the $create$ function, except that a matching counterpart o_p of the object o_s in the model skeleton already exists in the model repository. The pre-condition of this function is:

$$(owns(o_s) \wedge (owners(o_p) \setminus \{b\} \neq \emptyset)) \implies shared(o_p)$$

The pre-condition ensures that objects which cannot be shared between multiple agents can only be owned by a single agent. If the pre-condition is not fulfilled, the merging aborts with a conflict state. The $link$ function updates the $refs'$ and $owner'$ in the model repository m'_p accordingly. This is enforced by the following post-condition:

$$b \in refs'(o_p) \wedge (owns(o_s) \Leftrightarrow b \in owners'(o_p))$$

- $remove(o_p)$: This function removes the agent b from the list of referencing agents and removes the object from the model repository if the number of references is zero. Its post-condition is:

$$b \notin owners'(o_p) \wedge b \notin refs'(o_p) \wedge (refs'(o_p) = \emptyset \Leftrightarrow o_p \notin O'_p)$$

- $merge(o_s, o_p)$: This function synchronizes the contents of the object o_s in the model skeleton and its counterpart o_p in the model repository. We define $\Omega_{o_p} = \{f \in F \mid domain(F) \in (class(o_p) \cup superTypes(class(o_p)))\}$

that contains all features of the class of o_p as well as all its super classes. Then the pre-condition of this function is:

$$\forall f \in \Omega_{o_p} : (shared(o_p) \wedge f \notin id(class(o_p))) \Rightarrow (many(f) \wedge \neg ordered(f))$$

Single-valued or ordered multi-valued features are not permitted for objects which are shared between agents, as we may overwrite changes of other agents. Identifier features used for matching are excluded given that they are ensured to always have the same value in o_s and o_p. The function has two post-conditions depending on whether it is a single-valued or multi-valued feature:

$$\exists o'_p \in O'_p, \forall f \in \Omega_{o_p} : many(f) \Rightarrow fv_f(o'_p) = fv_f(o_p) \cup fv_f(o_s)$$

$$\exists o'_p \in O'_p, \forall f \in \Omega_{o_p} : \neg many(f) \vee ordered(f) \Rightarrow fv_f(o'_p) = fv_f(o_s)$$

In case of multi-valued, unordered features, we create the union of all its values in o_s and o_p. In case of single-value or an ordered multi-value features, we overwrite the value of the feature in the model repository with the one in the model skeleton.

The merging step calls the functions in the following order: a) $create$ for each object in set Δ_{new}, b) $link$ for each object in set Δ_{exists}, c) $remove$ for each object in set Δ_{remove}, and d) $merge$ for the set of objects $\Delta_{owned} = \{o_s \in O_s \mid owns(o_s)\}$.

Conflicts. If any of the pre-conditions of the merging primitives above were violated, the merging would fail in a conflict state. Given that we do not assume that a user may manually help to resolve the conflicts, we need to ensure that conflicts may not happen in a correctly set up system. The following invariants need to hold to ensure a conflict-free model repository:

- If a model object is allowed to be shared between agents, its non-identifying features may only be non-ordered and multi-valued.

$$\forall o_p \in O_p, \forall f \in \Omega_{o_p} : shared(o_p) \wedge f \notin id(class(o_p)) \implies many(f) \wedge \neg ordered(f)$$

- Each non-shared model object may be only owned by a single agent:

$$\forall o_p \in O_p : \neg shared(o_p) \implies |owners(o_p)| \leq 1$$

The first invariant can be enforced through a deliberate formulation of the $shared$ and id functions. The second invariant needs to be checked at system run-time. Our idea to avoid conflicts at run-time is to utilize the agent roles introduced in Section 2.3. We require each agent to specify on startup, which roles it fulfills. The agent is only permitted access to the model repository if in the same extraction scope no other agent with any of these roles is registered

or if a role explicitly allows multiple agents. Therefore, these variants can be enforced by the centralized monitoring repository utilizing the agent roles defined in Section 2.3. This way, no communication between the different agents is required.

We now discuss, which agent roles may allow multiple agents of the same role in the same extraction scope. Given agent role t, the subset $C_t \subseteq C$ of meta-model classes specifies which objects an agent in t may own in its model skeleton. It is important to note, that the agent still may reference objects from the full set C. If for all classes in C_t holds that all non-identifying features are non-ordered and multi-valued, multiple agents of role t may be allowed in the same extraction scope.

We determined the functions $shared$ and id for the DML meta-model and identified the following agent roles that allow for sharing of model objects: *D1* (Data Center Structure), *D5* (System Interface Providing Roles), *D6* (Application Assembly), *A10* (Component Boundary), *A2* (Component Assembly), *A3* (Component Deployment), and *A5* (Coarse-Grained Behavior).

3. Related Work

There exists a lot of work on the topic of performance model extraction. In order to group and relate them to our work, we cluster the existing approaches according to which agent role it could provide. We believe this improves the plausibility as well as the practical applicability of our proposed architecture. We follow the same structure as with the description of the agent roles in Section 2.3.

Data Center scope. Traditional system or network management software, such as IBM Tivoli or Hyperic, and virtualization management software, such as VMware vCenter, are commonly used to manage data centers. Such software typically maintains an inventory of the systems and applications as well as their topology in a data center. Proprietary or standardized management interfaces, such as Simple Network Management Protocol (SNMP), or Common Information Model (CIM), are available to access that information.

Usage Scope. Table 1 gives an overview of existing approaches to usage model extraction. Session logs can be collected either on the server or on the client. On the server side, access logs (e.g., on a web server) are often available containing the required information. In recent years, user monitoring on the client side becomes increasingly popular (e.g., using javascript instrumentation of web pages, such as Google Analytics). Numerous approaches to web mining (see, e.g., the work of Liu and Keselj [18]) have been proposed in the literature. However, such techniques are focused on the general analysis of user behavior.

Sharma et al. [19] uses Independent Component Analysis (ICA) to classify user requests according to their resource needs. However, the approach does not consider sessions consisting of several requests. Thus it only covers agent

Approach	U1	U2	U3
Web mining techniques [e.g. 18]	✗		
Sharma et al. [19]	(✗)		
CBMG [20]	✗		
WESSBAS [21]	✗		
LIMBO [22]		✗	
Brosig et al. [4]			✗

Table 1: Existing approaches to usage model extraction.

role *U1* partially. Menascé et al. [20] and van Hoorn et al. [23] consider the extraction of usage models for performance prediction. The work of Menascé et al. [20] proposes a modeling formalism called Customer Behavior Model Graph (CBMG) to describe user behaviors. They employ clustering techniques to determine different types of user sessions and reconstruct the usage behavior (see *U1*) by analyzing the sequence and timings of observed sessions in a session log. Van Hoorn et al. [23] published a tool, called WESSBAS, based on similar techniques to extract usage behaviors for load testing.

Kistowski et al. [22] propose an approach, called LIMBO, to extract models describing the temporal development of load intensities. The approach employs signal processing techniques (e.g., Fourier transformations) to identify seasonal patterns, trends, bursts, and noise. Brosig [24] considers supervised learning techniques to group values of input parameters according to their performance impact.

Platform Scope. The extraction of platform layers is highly technology-specific. We limit our discussion to hypervisors. Virtualization management software (e.g., VMware vCenter) provide access to the hypervisor configuration to well-documented interfaces. These interfaces are useful to implement *P1* agents. In addition, they typically also provide comprehensive monitoring capabilities supporting *P3* agents. However, they contain purely descriptive models without support for predictive analyses.

The extraction of hypervisor overheads (see role *P2* in Section 2.3.3) is an active research field. The work of Huber et al. [15] uses micro-benchmarks to determine the performance impact of certain hypervisor configurations. Lu et al. [25] employ directed factor graphs with regression analysis techniques in order to map the resource usage statistics observed within a VM to corresponding statistics at the hypervisor level including hypervisor overheads.

Application Scope. A broad set of static or dynamic analysis techniques are available for application model extraction. Table 2 gives an overview of existing approaches to application model extraction. Awad and Menascé [26] and Israr et al. [27] extract QNs and respectively, Layered Queueing Networks (LQNs) models. These models are not component-based, therefore they do not fulfill roles *A1*, *A2*, and *A3*. Awad and Menascé [26] only determine call frequencies, whereas Israr et al. [27] use traces of individual transactions to determine more fine-grained control flows.

Approach	A1	A2	A3	A4	A5	A6	A7
Awad and Menascé [26]					✗		
Israr et al. [27]				✗			
SoMoX [28]	✗	✗		✗		✗	
Brosig et al. [4]	✗	✗	✗	✗			
PMW [5]	✗	✗	✗	✗			
SLAstic [11]	✗	✗	✗		✗		
PMX [29]	✗	✗	✗	✗			

Table 2: Existing approaches to application model extraction

SoMoX [28] uses a combination of static and dynamic analysis techniques to extract PCM instances. It requires access to the source code of an application and uses clustering techniques on code metrics to identify components as well as the component assembly. In order to determine parameter dependencies, the application is executed in a dedicated test environment and genetic search techniques are employed. Given that the approach is targeted at model extraction at design time, it lacks the information where the application will be deployed at run-time.

Brosig et al. [4], PMW [5], SLAstic [11] and PMX [29] are exclusively based on dynamic analysis techniques of applications. Brosig et al. [4], PMW [5] and PMX [29] are focused on the extraction of PCM instances, while SLAstic uses its own meta-model (although a transformation to PCM exists). The main difference between the four approaches is the monitoring tools used for obtaining the input for the dynamic analysis. Brosig et al. [4] is based on the proprietary instrumentation techniques provided by the Oracle WebLogic middleware platform. PMW uses standardized Java EE filters to intercept incoming requests or alternatively can exploit session data from the Dynatrace Application Performance Management (APM) tool[2]. SLAstic and PMX are based on the Kieker application monitoring framework [30].

Model Variable Scopes. Table 3 shows major existing approaches to model parameterization that may be used to implement the agents described in Section 2.3.5. The works of Courtois and Woodside [31] and Westermann et al. [32] propose regression techniques to determine functions on the observed response time. LibReDE [33] is a tool for resource demand estimation, which has also been integrated with the PMW [5] and the PMX [29] tools. Brunnert et al. [5] also implement a measurement-based approach for resource demands using application instrumentation. Further approaches using resource demand estimation techniques are Wang et al. [34] and Brosig et al. [4]. The ByCounter [35] approach uses fine-grained instrumentation to count byte-code instructions and micro-benchmarks to measure the resource demand of individual instructions.

Israr et al. [27], Brosig et al. [4], PMW [5], PMX [29] and SLAstic [11] are all using dynamic analysis techniques to determine the control flow of appli-

[2] `http://www.dynatrace.com/de/index.html`

Approach	M1	M2	M3	M4	M5
Courtois and Woodside [31]	✗				
Westermann et al. [32]	✗				
LibReDE [33]		E			
Wang et al. [34]		E			
Israr et al. [27]			✗		
SoMoX [28]			✗	✗	✗
ByCounter [35]		M			
Brosig et al. [4]		E	✗	✗	✗
PMW + LibReDE [5]		M/E	✗		
SLAstic [11]			✗		
PMX + LibReDE [29]		E	✗		

Table 3: Existing approaches to model parameterization (E stands for estimation and M for measurement).

cations, including a characterization of control flow variables. In contrast, SoMoX [28] uses static analysis techniques to reach that goal. Combined with dynamic analysis techniques, SoMoX can also characterize parameter dependencies using explicit stochastic expressions. Brosig et al. [4] characterize parameter dependencies using empirical distributions.

4. Aspects of Agent Implementations

In this section, we describe our reference implementation of three different model learning agents. The agents automatically extract submodels describing the structure and configuration of the data-center infrastructure as well as the control flows, deployments, and the parameterization of the application architecture. The agents continuously send the submodels to the performance model repository. Using the merging algorithm presented in Section 2.5, they are automatically merged into the existing performance model. This enables the automatic extraction of up-to-date performance models from a running application.

4.1. VMware vSphere Agent

The vSphere Agent described in this section implements the agent roles Data Center Structure (D1), Compute Node Configuration, Platform Configuration (P1) and Platform Monitoring (P3).

VMware vSphere is a virtualization platform supporting the centralized management of clusters of x86 servers. We chose vSphere due to its widespread use in industry [36] within public and private cloud infrastructures. Our model extraction agent builds upon the public web service interface of vSphere to extract sub-models for the data center scope and platform scopes. The agent itself runs in a system VM with access to the management network of vSphere.

The vSphere platform consists of the ESX hypervisor and the vCenter server for hypervisor management in a cluster of virtualized hosts. A vCenter server

manages an inventory of the physical hardware, the hypervisor configuration and the VMs deployed in a cluster. Furthermore, the vCenter server collects and stores detailed monitoring statistics from all hypervisor instances. The agent accesses this information through web services provided by the vCenter server (see vendor's documentation [37]). The observed structure is then automatically mapped and translated into a valid DML model.

In order to reflect changes to the system at run-time, clients can register at a vCenter server to be automatically notified of changes in the inventory. This notification mechanism covers manual reconfigurations of a system administrator as well as any inventory changes from the system itself. Our agent registers for any changes that need to be reflected in the DML resource landscape model. Each notification contains a pointer to the changed objects so that only the corresponding subset of the DML model needs to be updated.

4.2. JavaEE Wildfly Agent

This section describes the implementation of the Wildfly agent, which fulfills agent roles A1 (Component Boundary), A4 (Fine-Grained Behavior), A2 (Component Assembly), A8 (Application Monitoring), and A3 (Component Deployment).

We built a virtual appliance based on a CentOS 6 Linux operating system, an OpenJDK 7 Java VM, and the Wildfly 8.2 application server. Wildfly (formerly known as JBoss) is an open-source application server[3] fully compliant with the Java EE standard. It is written in Java and runs on any standard Java VM. Several Wildfly instances can form a cluster to fulfill high-availability goals using replication and load-balancing techniques. Our agent is configured to support clustering and runs in a domain mode for easier cluster management.

On startup of a Wildfly node, it is not yet clear which application components will be deployed on this instance. Furthermore, the deployment of components may change dynamically during system run-time. Therefore, our agents determine at run-time which components are deployed on a server, as well as the control flow between these components. The latter requires the insertion of instrumentation points at providing and requiring interface roles to observe the inwards and outwards flow of requests. We developed a custom agent for the Wildfly server that is loaded directly into the server process and that has full access to the current server state. The agent automatically intercepts all deployments of components and inserts the required instrumentation points.

In the following, we describe the static analysis steps that are performed by the agent when new components are deployed, and then we give an overview of the dynamic analysis steps implemented in our agent.

[3] http://wildfly.org/

4.2.1. Static Analysis

When the component deployment on a Wildfly node changes, our model extraction logic needs to be informed of these changes. Wildfly offers an extension point to provide custom deployment unit processors that are invoked when application modules are added or removed. Custom processors need to implement the `org.jboss.as.server.deployment.DeploymentUnitProcessor` interface. We provide an implementation that performs the following steps for each application module:

- *Static analysis of component structure:* It determines the components contained in an application module as well as their type (i.e., web or Enterprise Java Bean (EJB) components). For each component, it identifies the interface providing ports.
- *Instrumentation setup:* It adds instrumentation points to observe incoming and outgoing invocations of a component. This information is required for the dynamic analysis.

Interface requiring roles are difficult to determine statically. Java EE provides two different ways to obtain references to required components: *dependency injection* and *Java Naming and Directory Interface (JNDI) lookup*. In the former case, the container already knows all required components at deploy time. However, in the latter case, an analysis of the complete bytecode of a component would be required to find all JNDI lookups of required components. To avoid a full bytecode analysis, we resort to dynamic analysis techniques to determine the interface requiring ports of components.

4.2.2. Dynamic Analysis

The interceptors count each incoming or outgoing invocation and measure its execution time. The agent does not maintain statistics for each individual invocation in order to keep the volume of monitoring data low. Instead, it accumulates the values for each component service and each external call within a component. In regular intervals (e.g., every minute), it sends the aggregated statistics to the performance model repository. If the agent detects a component service or an external call that has not been observed before, it triggers an update of the model skeleton. The update is performed asynchronously to avoid delaying the application processing.

Each new component service or external call is added to the model skeleton initially created when performing the static analysis of the component structure during deployment. In case of external calls, the agent automatically adds assembly connectors between source and target component instances if required. It uses technical identifiers (e.g., Uniform Resource Locators (URLs)) as used by the Wildfly server internally to identify components in a unique way. The assumption is that it can always determine the target component instance on the client-side. Updates to the model skeleton are sent in batches to the performance model repository in order to reduce communication overhead.

4.3. Librede Agent

In order to provide the remaining mandatory agent roles M2 (Resource Demand) and M3 (Control Flow), we introduce the Librede agent. The statistics collected by the Wildfly agent described in the previous section are now forwarded to the Library for Resource Demand Estimation (LibReDE) Agent in order to classify the resource demands. LibReDE is a library of ready-to-use implementations of state-of-the-art approaches to resource demand estimation that can be used for online and offline analysis [33]. It uses several different statistical estimation approaches to estimate the resource demands based on generic system- and application-level measurements provided by the other agents. We evaluated all available approaches and chose the available Recursive optimization using response times and utilization observations [38] for this case study.

5. Evaluation

We evaluate the results of our work in this section. For evaluation, we choose the SPECjEnterprise2010 benchmark, which is described in Section 5.1. After that, we evaluate two aspects of our reference architecture: the degree of automation in Section 5.2 and the predictive power of the resulting model in Section 5.3.

5.1. Experiment Setup

SPECjEnterprise2010 is an industry-standard full system benchmark for Java EE application servers[4]. The goal of the benchmark is to enable the comparison of different Java EE application servers with regards to their scalability and their efficiency under real-world applications. It covers the full system stack and uses an application workload representative of many real-world enterprise systems. The benchmark is designed to exploit a large set of different Java EE 5 technologies covering dynamic web pages (Servlets and Java Server Pages), web services, (distributed) transactional EJBs, asynchronous messaging (Java Messaging Service) and object persistence (Java Persistence API).

In this case study, we evaluate the degree of automation and the prediction accuracy of the models obtained using our reference architecture for online performance model extraction presented in this paper. We chose the SPECjEnterprise2010 benchmark as it provides a complex workload representative of many real-world enterprise applications and exploits a broad set of technologies of

[4]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 website for SPECjEnterprise2010 is located at http://www.spec.org/jEnterprise2010.

the Java EE standard. These properties make the benchmark also an ideal candidate to evaluate the capabilities of approaches for performance model learning. As a result, it has become a de-facto benchmark in this research area for both industry [39] and academia [e.g., 4, 5, 40, 41, 42].

Workload. The benchmark workload consists of Customer Relationship Management (CRM), manufacturing and supply-chain management applications. The business scenario of the benchmark is modeled after an automotive manufacturer with car dealerships, manufacturing sites, and suppliers interacting with the system. Car dealerships use interactive web applications to access the order domain where they can browse, purchase and sell cars. The manufacturing sites use remote EJB and web service calls to start and complete manufacturing processes in the manufacturing domain. Suppliers are triggered through a web service interface by the supplier domain if parts need to be purchased for manufacturing.

SPECjEnterprise2010 comes with two workload drivers: one for generating workloads from car dealerships (*DealerDriver*) and the other for the manufacturing sites (*MfgDriver*). The generated workloads are based on transactions; each transaction sending a sequence of different requests to the system. Car dealerships interact with the order domain using either *Browse*, *Purchase* or *Manage* transactions. The *Browse* transaction is dominated by read requests, whereas the latter two transactions are a mixture of read and write requests. The manufacturing sites communicate either through a Simple Object Access Protocol (SOAP)-based web service or through binary Remote Method Invocation (RMI)-based protocols. We distinguish between *Mfg WS* and *Mfg EJB* transactions accordingly. The sequence of requests in a transaction is defined by a first-order Markov chain. We use the standard workloads as defined in the standard [43]. The workload drivers can be configured with a transaction rate which determines the number of concurrent threads in the load driver sending requests to the system. The transaction rate scales the interarrival times of the different types of transactions accordingly.

The external suppliers are represented by one or multiple emulators, which wait for requests from the supplier domain. It simulates the processing of purchase orders for components required to manufacture a car in the manufacturing domain. After receiving a purchase order, it sleeps for a certain time defined by the lead time of the requested component. Then it signals the shipment of the component to the supplier domain.

Deployment. The benchmark is originally designed for a three-tier deployment, consisting of a web, an application, and a database server. In addition, the three domains may be deployed on separate servers. However, modern enterprise applications often follow a service-oriented paradigm implementing the functionality as multiple independent services that can be deployed separately. In order to better reflect the architecture of a service-oriented, distributed system, we adapted the SPECjEnterprise2010 benchmark, so that the EJBs in the business logic tier can be deployed individually as services. Figure 9 shows the

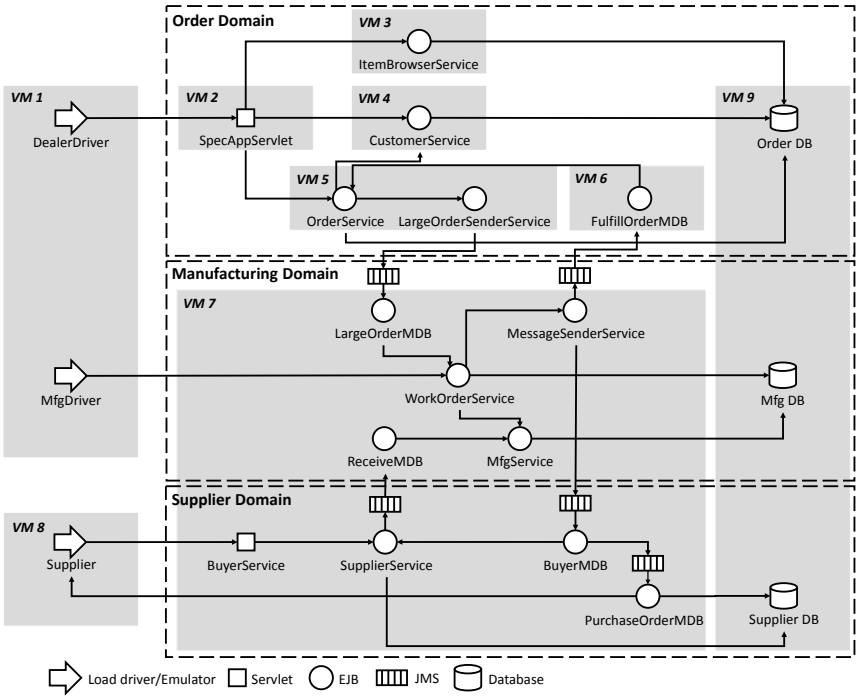

Figure 9: Distributed deployment of SPECjEnterprise2010.

resulting deployment. The benchmark is deployed on a cluster of Wildfly 8.2 application servers. The order domain is distributed over several fine-granular services each deployed in a separate virtual machine. The services of the manufacturing and supplier domains are all deployed in the same VM. The data tier is shared by all business services and hosted by a MySQL 5.6.25 relational database. The communication between business services is based on the RMI protocol as provided by the application server. The relational database is accessed using the standard Java Database Connectivity (JDBC) drivers provided by MySQL.

The physical resource environment consists of 4 servers, each equipped with 1 Intel Xeon E3-1230 CPU with 4 cores, 16 GB main memory, 500 GB HDD and 1 Gbit network connection. VMware vSphere 5.5 is used as hypervisor. All VMs are equipped 1 virtual CPU and 4 GB memory, except for VM 7 (2 vCPUs and 8 GB memory) and VM 9 (2 vCPUs and 4 GB memory) which have a higher resource requirement. The VMs are distributed evenly between hosts so that resources are not over-committed (Host 1: VM 1, VM 8; Host 2: VM 3, VM 5, VM 6; Host 3: VM 2, VM 7; Host 4: VM 9, VM 4). Each VM runs a CentOS 6.6 Linux 64-bit operating system.

Experiment Runs. The DML model used for evaluation purposes was automatically extracted by our approach during an eight-hour benchmark run at a trans-

action rate of 60 using our agents described in Section 4. Both our agent implementations and the extracted model are publicly available for download[5]. For validation purposes, we performed five more benchmark runs each with a duration of one hour varying the transaction rates between 20 (corresponds to a lightly utilized system) and 100 (which is close to the maximum sustainable load at VM 3).

5.2. Degree of Automation

The following agent roles (see Section 2.3) are automated in this case study: (D1), (P1), (P3), (A1), (A2), (A3), (A4), (A8), (M2) and (M3). The individual sub-models created by these agent roles could be autonomously merged without conflicts as the invariants documented in Section 2.5 are fulfilled. In order to obtain a complete DML model, we performed additional manual steps in this case study. The agent roles *System Interface Providing Roles* (D5), *Application Assembly* (D6), and all agent roles in the usage scope (U1), (U2), (U3) were not covered by automated agents and had to be created manually.

Sub-Model	Total Elements	Manually Created Elements	Degree of Automation
UsageProfile	208	207	0.5 %
System	12	11	8.3 %
ResourceLandscape	60	0	100.0 %
Deployment	15	0	100.0 %
Repository			
Structure	316	10	96.8 %
Behavior	584	14	97.6 %
Parameterization	543	12	97.8 %
Total	1739	254	85.4 %

Table 4: Analysis of manual effort required for model creation

To quantify the achieved degree of automation, we compare the number of manually and automatically created model elements. In DML all model entities have a similar level of complexity since complex elements are modeled as a composition of multiple model entities. Consequently, the basic model elements counted during this evaluation all have a similar level complexity. This is an inherent property of the Eclipse Modeling Framework (EMF), in which DML is implemented. Therefore, percentage of automatically created model elements is a suitable metric to quantify the degree of automation.

[5]See `http://descartes.tools/prisma` for the implementation of the reference architecture along with a list of all available agents. The model we used in this evaluation can be found at `https://gitlab2.informatik.uni-wuerzburg.de/descartes/prisma-core/tree/master/examples/specjenterprise2010`.

Table 4 depicts the degree of automation achieved for the DML model of the SPECjEnterprise2010 benchmark extracted during our experiments. The resulting model consists of six submodels containing a total of 1739 model entities. Out of these model entities 254 were created manually, resulting in an overall degree of automation of 85.4%. In the following, we discuss how the current limitations can be relieved to reach the goal of full automation:

- The manual effort for creating the usage profile makes up for 207 out of 254 manually created elements. In Section 2.3.2, we list several techniques that can be used here. For instance, van Hoorn et al. [23] propose a technique to automatically extract usage profiles that they have already successfully validated with the SPECjEnterprise2010 workload [21]. We leave the integration of such techniques into our reference architecture as future work.

- The emulator is not part of the extraction as it represents an external component, which may be hosted outside of the data center. For performance prediction purposes, we manually added a component representing the emulator in our model. The emulator accounts for 35 out of the 36 manually created repository model entities. Future work should consider the automatic extraction of more appropriate models for external services.

- The overall application assembly, i.e., the applications in a data center and the communication paths between applications need to be defined manually. The monitoring of the control flow across different applications in a data center is an open challenge. Today's monitoring tools for applications are mostly focused on single applications. However, a system administrator typically needs to know the high-level control flow of an application (i.e., the externally provided and required services) anyways to configure the system correctly. Therefore, we think that it would be an acceptable manual effort. In our case study, the manual effort for the application assembly only makes up 13 model entities and therefore 0.7% of all model entities.

Manual parts can be easily integrated into the end-to-end performance model by creating model skeletons by hand. As model skeletons are valid instances of DML, the existing Eclipse tooling for the graphical and textual editing of DML models [24] can be used to create them. They can be uploaded to our model repository through the same interface used by the model extraction agents.

5.3. Model Prediction Accuracy

In this section, we evaluate the prediction accuracy of the DML model extracted in our case study under different transaction rates. To evaluate the prediction accuracy for scaling decisions, we now use our extracted model and compare the predicted utilization and end-to-end response times at the transaction levels 20, 40, 60, 80, and 100 with measurements from corresponding benchmark runs at the real system. The transaction levels are chosen to cover

a wide range of resource utilizations. A transaction level of 100 is close to the maximum load sustainable with the given configuration. Higher transaction rates require additional resources to ensure system stability.

The results are shown in Figure 10 for the CPU utilization and in Figure 11 for the end-to-end response time. The response times are shown for complete transactions consisting of multiple individual requests to the system.

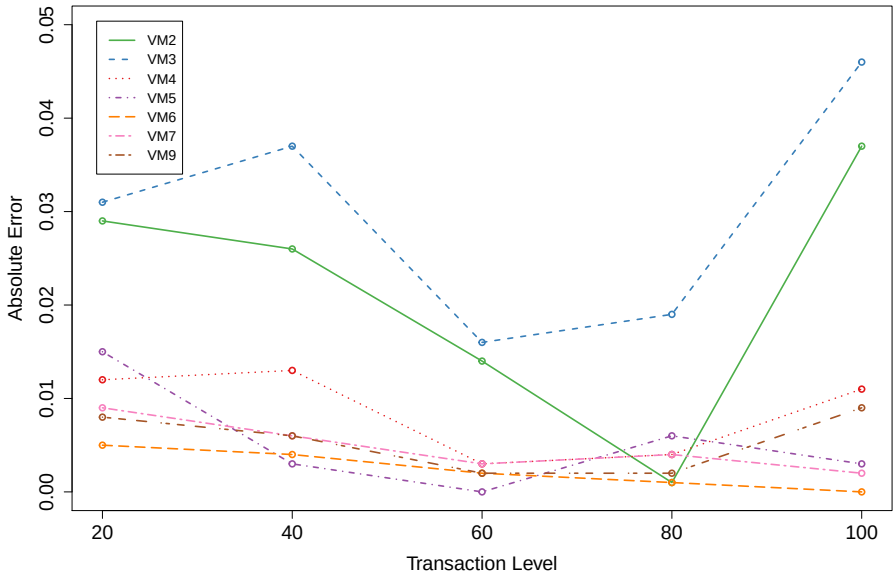

Figure 10: Mean absolute utilization error for different transaction rates.

Figure 10 shows the mean absolute prediction error for the different VMs. The deployment of the VMs is discussed in Figure 9. Generally speaking, the utilization error is always below 5%. For VM6, VM7, and VM9, the error is even below 1%. This is very positive since VM7 runs a majority of our services. In terms of general utilization, VM2 and VM3 were usually the ones with the highest utilization in the system. Therefore, their higher prediction error is acceptable, since the graph shows the absolute utilization errors. However, no general trend about higher transaction rates influencing our prediction can be observed.

Figure 11 depicts the relative end-to-end response time error of our predictions. We can observe a much higher variation of the errors when varying the transaction rate. Generally, the transactions Purchase, Browse and also Manage seem to be harder to predict than both transactions concerning the manufacturing (Mfg EJB and Mfg WS). However, at a transaction rate of 60, the relative errors of all considered transactions drop below 5%. Interestingly, although the Browse transaction is dominated by read requests (as opposed to Purchase and

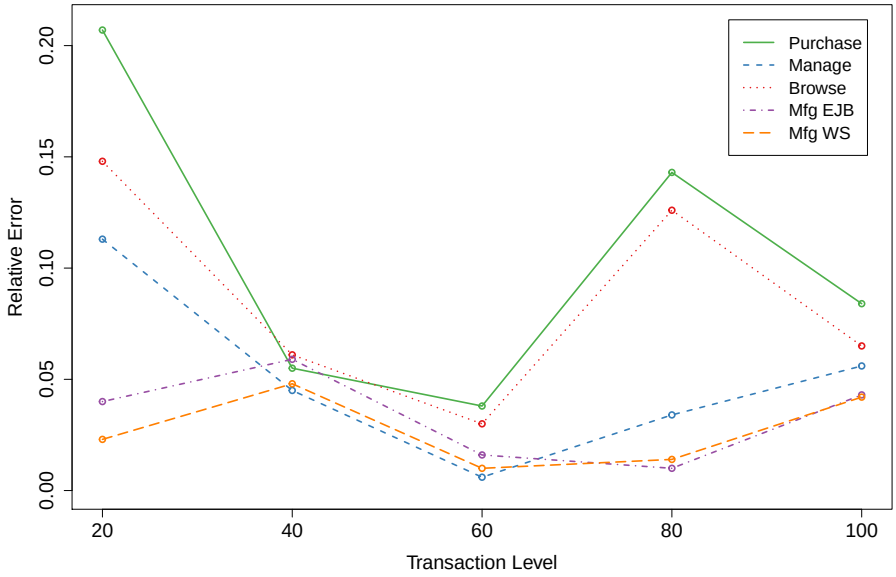

Figure 11: Mean relative end-to-end response time error for different transaction rates.

Manage), this does not seem to affect the prediction capabilities of our system.

In summary, the extracted DML model yields a high prediction accuracy. The absolute errors of the utilization are all below 4% and the relative errors of the end-to-end response times are less than 21%. A prediction error of 30% concerning mean response times and 5% concerning resource utilization is considered acceptable in capacity planning [44].

6. Summary

In this work, we presented a reference architecture for online model learning in virtualized environments. The reference architecture is based on *agents* which are responsible for extracting model skeletons of certain aspects of a system. Agents may employ different *static* and *dynamic* analysis techniques to create model skeletons at run-time. We expect a *deep integration* of agents into existing technologies and platforms in order to exploit domain-specific knowledge for model learning. The model skeletons are *dynamically composed* into a comprehensive performance model of a system. Our reference architecture is based on a common meta-model in order to ease the composition of model skeletons from different agents.

In order to implement our reference architecture, virtualization platforms need to be extended with additional components supporting the online model learning. However, these components are supplementary and do not require

changes in the existing parts of a virtualization platform. We provide a reference implementation of the core components.

We evaluated the degree of automation achieved for the model learning step as well as the fitting and prediction accuracy of the resulting performance models. For the SPECjEnterprise2010 application, we achieved a degree of automation of 85.4%. By integrating existing techniques for usage profile extraction, this number could be increased to 97.3%. As a result, a system administrator only needs to provide high-level information on the control flow between applications to obtain a complete model. The internal architecture of the application could be completely extracted including all platform layers. It is noteworthy, that the model learning capabilities in the Wildfly VAs only assume that the application adheres to the Java EE standard and does not include any prior knowledge of the SPECjEnterprise2010 application. Therefore, it can be reused to create performance models of other Java EE applications. For instance, [45] has already used our Wildfly VA with a different application.

We then use the fully parameterized model to predict the performance for scaling scenarios. We obtained model predictions with an absolute error of less than 4% for the CPU utilization and a relative error of less than 21% for the end-to-end response time. Given the SPECjEnterprise2010 deployment with seven different VMs and 80 different resource demands to be estimated, it poses a considerable problem size.

Future work. Leveraging our reference architecture, future work may provide VAs containing model extraction agents focused on specific aspects of model learning. A performance engineer, who has expertise in performance modeling, can specifically design the extraction logic to exploit a priori knowledge about a technology and leverage proprietary interfaces. For a given technology, this needs to be done only once and the resulting VA can be reused in different deployments.

Acknowledgments

This work was supported by the German Research Foundation (DFG) under grant No. (KO 3445/11-1).

References

[1] B. Jennings, R. Stadler, Resource management in clouds: Survey and research challenges, J. Network Syst. Manage. 23 (3) (2015) 567–619.

[2] T. Lorido-Botran, J. Miguel-Alonso, J. A. Lozano, A review of auto-scaling techniques for elastic applications in cloud environments, J. Grid Comput. 12 (4) (2014) 559–592.

[3] N. Huber, F. Brosig, S. Spinner, S. Kounev, M. Bähr, Model-based self-aware performance and resource management using the descartes modeling language, IEEE Transactions on Software Engineering 43 (5) (2017) 432–452.

[4] F. Brosig, N. Huber, S. Kounev, Automated extraction of architecture-level performance models of distributed component-based systems, in: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, ASE, 2011, pp. 183–192.

[5] A. Brunnert, C. Vögele, H. Krcmar, Automatic performance model generation for java enterprise edition (EE) applications, in: Proceedings of the 10th European Workshop Computer Performance Engineering, EPEW, 2013, pp. 74–88.

[6] S. Spinner, G. Casale, F. Brosig, S. Kounev, Evaluating Approaches to Resource Demand Estimation, Elsevier Performance Evaluation 92 (2015) 51 – 71.

[7] S. Kounev, N. Huber, F. Brosig, X. Zhu, A Model-Based Approach to Designing Self-Aware IT Systems and Infrastructures, IEEE Computer 49 (7) (2016) 53–61.

[8] F. Brosig, N. Huber, S. Kounev, Architecture-level software performance abstractions for online performance prediction, Sci. Comput. Program. 90 (2014) 71–92.

[9] N. Huber, A. van Hoorn, A. Koziolek, F. Brosig, S. Kounev, Modeling run-time adaptation at the system architecture level in dynamic service-oriented environments, Service Oriented Computing and Applications 8 (1) (2014) 73–89.

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

[11] A. van Hoorn, Model-driven online capacity management for component-based software systems, Ph.D. thesis, Faculty of Engineering, Kiel University, Kiel, Germany (2014).

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

[13] Q. Noorshams, Modeling and prediction of i/o performance in virtualized environments, Ph.D. thesis, Karlsruhe Institute of Technology (KIT) (2015).

[14] P. Rygielski, S. Kounev, Descartes Network Infrastructures (DNI) Manual: Meta-models, Transformations, Examples, Technical Report v.0.3, Chair of Software Engineering, University of Würzburg, Am Hubland, 97074 Würzburg (2014).

[15] N. Huber, M. von Quast, M. Hauck, S. Kounev, Evaluating and modeling virtualization performance overhead for cloud environments, in: CLOSER 2011 - Proceedings of the 1st International Conference on Cloud Computing and Services Science, 2011, pp. 563–573.

[16] B. Westfechtel, Merging of EMF models - formal foundations, Software and System Modeling 13 (2) (2014) 757–788.

[17] S. Förtsch, B. Westfechtel, Differencing and merging of software diagrams - state of the art and challenges, in: Proceedings of the Second International Conference on Software and Data Technologies - Volume 2: ICSOFT,, INSTICC, SciTePress, 2007, pp. 90–99.

[18] H. Liu, V. Keselj, Combined mining of web server logs and web contents for classifying user navigation patterns and predicting users' future requests, Data Knowl. Eng. 61 (2) (2007) 304–330.

[19] A. B. Sharma, R. Bhagwan, M. Choudhury, L. Golubchik, R. Govindan, G. M. Voelker, Automatic request categorization in internet services, SIGMETRICS Performance Evaluation Review 36 (2) (2008) 16–25.

[20] D. A. Menascé, V. Almeida, R. C. Fonseca, M. A. Mendes, A methodology for workload characterization of e-commerce sites, in: EC, 1999, pp. 119–128.

[21] A. van Hoorn, C. Vögele, E. Schulz, W. Hasselbring, H. Krcmar, Automatic extraction of probabilistic workload specifications for load testing session-based application systems, EAI Endorsed Trans. Self-Adaptive Systems 1 (3) (2015) e5.

[22] J. von Kistowski, N. R. Herbst, D. Zöller, S. Kounev, A. Hotho, Modeling and extracting load intensity profiles, in: 10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, 2015, pp. 109–119.

[23] A. van Hoorn, C. Vögele, E. Schulz, W. Hasselbring, H. Krcmar, Automatic extraction of probabilistic workload specifications for load testing session-based application systems, in: 8th International Conference on Performance Evaluation Methodologies and Tools, 2014, pp. 139–146.

[24] F. Brosig, Architecture-level software performance models for online performance prediction, Ph.D. thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany (2014).

[25] L. Lu, H. Zhang, G. Jiang, H. Chen, K. Yoshihira, E. Smirni, Untangling mixed information to calibrate resource utilization in virtual machines, in: Proceedings of the 8th International Conference on Autonomic Computing, 2011, pp. 151–160.

[26] M. Awad, D. A. Menascé, Dynamic derivation of analytical performance models in autonomic computing environments, in: Proceedings of the 2014 Computer Measurement Group Conference, 2014, pp. 159–168.

[27] T. A. Israr, C. M. Woodside, G. Franks, Interaction tree algorithms to extract effective architecture and layered performance models from traces, Journal of Systems and Software 80 (4) (2007) 474–492.

[28] K. Krogmann, Reconstruction of software component architectures and behaviour models using static and dynamic analysis, Ph.D. thesis, Karlsruhe Institute of Technology (2010).

[29] J. Walter, Website, online available at `http://descartes.tools/pmx/`. Last accessed on 23-05-2016 (2015).

[30] M. Rohr, A. van Hoorn, S. Giesecke, J. Matevska, W. Hasselbring, S. Alekseev, Trace-context sensitive performance profiling for enterprise software applications, in: Performance Evaluation: Metrics, Models and Benchmarks, 2008, pp. 283–302.

[31] M. Courtois, C. M. Woodside, Using regression splines for software performance analysis, in: Workshop on Software and Performance, 2000, pp. 105–114.

[32] D. Westermann, J. Happe, R. Krebs, R. Farahbod, Automated inference of goal-oriented performance prediction functions, in: IEEE/ACM International Conference on Automated Software Engineering, 2012, pp. 190–199.

[33] S. Spinner, G. Casale, X. Zhu, S. Kounev, LibReDE: A Library for Resource Demand Estimation, in: Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering, ACM Press, New York, NY, USA, 2014, pp. 227–228.

[34] W. Wang, J. F. Pérez, G. Casale, Filling the gap: a tool to automate parameter estimation for software performance models, in: Proceedings of the 1st International Workshop on Quality-Aware DevOps, 2015, pp. 31–32.

[35] M. Kuperberg, M. Krogmann, R. Reussner, ByCounter: Portable runtime counting of bytecode instructions and method invocations, in: Proceedings of the 3rd International Workshop on Bytecode Semantics, Verification, Analysis and Transformation, 2008.

[36] T. J. Bittman, P. Dawson, M. Warrilow, Magic quadrant for x86 server virtualization infrastructure (2016).

[37] VMware, Inc., vSphere API and SDK Documentation, Website, online available at https://pubs.vmware.com/vsphere-55/index.jsp. Last accessed on 04-02-2017 (2013).

[38] Z. Liu, L. Wynter, C. H. Xia, F. Zhang, Parameter inference of queueing models for IT systems using end-to-end measurements, Perform. Eval. 63 (1) (2006) 36–60.

[39] Standard Performance Evaluation Corporation (SPEC), Published SPECjEnterprise2010 Results, Website, online available at https://www.spec.org/jEnterprise2010/results/jEnterprise2010.html. Last accessed on 23-01-2018.

[40] F. Willnecker, H. Krcmar, Optimization of deployment topologies for distributed enterprise applications, in: 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA), 2016, pp. 106–115.

[41] C. Vögele, A. van Hoorn, E. Schulz, W. Hasselbring, H. Krcmar, Wessbas: extraction of probabilistic workload specifications for load testing and performance prediction—a model-driven approach for session-based application systems, Software & Systems Modeling 16 (2016) 1–35.

[42] A. Brunnert, H. Krcmar, Continuous performance evaluation and capacity planning using resource profiles for enterprise applications, Journal of Systems and Software 123 (2017) 239–262.

[43] Standard Performance Evaluation Corporation, SPECjEnterprise2010 Design Document, Website, online available at https://www.spec.org/jEnterprise2010/docs/DesignDocumentation.html. Last accessed on 02-02-2017 (2010).

[44] D. A. Menasce, A. F. A. Virgilio, Scaling for E Business: Technologies, Models, Performance, and Capacity Planning, 1st Edition, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[45] A. Bauer, Design and Evaluation of a Proactive, Application-Aware Elasticity Mechanism, Master Thesis, University of Würzburg, Am Hubland, Informatikgebäude, 97074 Würzburg, Germany (2016).