A Reference Architecture for Online Performance Model Extraction in Virtualized Environments

Conference Paper · March 2016

DOI: 10.1145/2859889.2859893

3 authors:

Simon Spinner
University of Wuerzburg
15 PUBLICATIONS **46** CITATIONS
SEE PROFILE

Jürgen Walter
University of Wuerzburg
5 PUBLICATIONS **2** CITATIONS
SEE PROFILE

Samuel Kounev
University of Wuerzburg
166 PUBLICATIONS **1,461** CITATIONS
SEE PROFILE

A Reference Architecture for Online Performance Model Extraction in Virtualized Environments

Simon Spinner
University of Würzburg
Am Hubland
97074 Würzburg, Germany
simon.spinner@uni-wuerzburg.de

Jürgen Walter
University of Würzburg
Am Hubland
97074 Würzburg, Germany
juergen.walter@uni-wuerzburg.de

Samuel Kounev
University of Würzburg
Am Hubland
97074 Würzburg, Germany
samuel.kounev@uni-wuerzburg.de

ABSTRACT

Performance models can support decisions throughout the life-cycle of a software system. However, the manual construction of such performance models is a complex and time-consuming task requiring deep system knowledge. Therefore, automatic approaches for creating and updating performance models of a running system are necessary. Existing work focuses on single aspects of model extraction or proposes approaches specifically designed for a certain technology stack. In virtualized environments, we often see different applications based on diverse technology stacks sharing the same infrastructure. In order to enable online performance model extraction in such environments, we describe a new reference architecture for integrating different specialized model extraction solutions.

Keywords

Architecture-level Performance Model; Model Extraction; Model Learning

1. INTRODUCTION

Performance models are an abstraction of a combined hardware and software system describing its performance-relevant structure and behavior. These models can be analyzed using analytical or simulation techniques providing predictions of the system performance in a given scenario. During a system's life-cycle, many different questions arise where performance predictions help to find better answers. For instance, performance models can be used during system design to choose between design alternatives [8], during system deployment to size a system for the expected workload [10] and during system operation to dynamically adapt the resource allocation to ensure a good system performance [7]. While performance models can provide many benefits, their manual creation and maintenance is time-consuming and expensive, severely limiting their usage in real-world systems.

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 [2, 4], or focuses on improving certain aspects of it (e.g., resource demand estimation [12]). In virtualized environments, multiple applications with diverse technology stacks typically share the same underlying infrastructure influencing each other. As a result, a performance model needs to represent the complete virtualized system (including the different applications) integrating information from heterogeneous datasources in order to enable reliable performance predictions. Furthermore, the deployment and configuration of applications may change frequently due to automatic or manual reconfigurations (e.g., deployment of new virtual machines (VMs), or migration of existing ones). As a result, the overall performance model of the system needs to be dynamically composed and continuously updated to reflect the current system state.

In this paper, we describe a new agent-based reference architecture for online performance model extraction in virtualized environments. The goals of this reference architecture are: (a) to enable the creation of purpose built agents focusing on specific model extraction tasks, (b) to simplify the reuse of general tools and algorithms for model extraction, (c) to enable the dynamic composition and parameterization of sub-models (called model skeletons), and (d) to allow the encapsulation of technology-specific knowledge in the agents. The agents may be bundled within VMs alongside the applications (as outlined in [14]), or may run in dedicated VMs. The agents will discover each other at run-time and collaborate to create a complete and fully parameterized performance model of the system.

The rest of the paper is organized as follows. Section 2 surveys the state-of-the-art in model extraction. Section 3 provides an introduction to the modeling formalism used in this paper. Section 4 gives an overview of the proposed reference architecture and Section 5 describes possible implementations of it. Section 6 concludes the paper.

2. STATE-OF-THE-ART

Existing approaches consider the problem of model extraction for analytical performance models [6, 1] and for

ICPE'16 Companion, March 12-18, 2016, Delft, Netherlands

DOI: http://dx.doi.org/10.1145/2859889.2859893

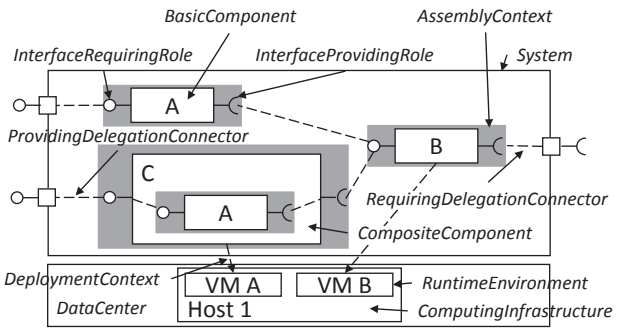

Figure 1: DML overview.

architecture-level performance models [9, 2, 4]. Hrischuk et al. [6] focus on generating Layered Queueing Network (LQN) models from traces collected by monitoring tools. However, their approach expects characterizations of certain model variables as input (e.g., resource demands). Awad and Menascé [1] derive analytical Queueing Network (QN) models dynamically. They propose a framework for automatic model identification, however, it can only provide rather coarse-grained models of the software architecture.

Krogmann [9] uses a combination of static and dynamic analyses to generate Palladio Component Model (PCM) instances from an existing application. The approach is aimed at reverse engineering tasks and does not support continuous model extraction in production systems. Brosig et al. [2] and Brunnert et al. [4] both describe extraction approaches for PCM tailored for Java Enterprise Edition application servers. However, they do not consider virtualized environments with heterogeneous software stacks and frequent reconfigurations.

Other work focuses on improving specific aspects of performance model extraction. Different statistical techniques for estimation resource demands based on monitoring data have been proposed (see the survey and experimental comparison by Spinner et al. [12]). Menascé et al. [11] and van Horn et al. [15] propose techniques to determine user behavior models. Herbst et al. [5] give an overview of state-of-the-art workload forecasting techniques to predict the load intensity over time. These techniques are supplementary to our work and can be integrated into our reference architecture.

3. DESCARTES MODELING LANGUAGE

For our reference architecture, we first need to decide on a performance modeling formalism. We use the Descartes Modeling Language (DML) [3, 7], because it is a descriptive, architecture-level performance model specifically targeted at online performance and resource management in data centers and it offers flexible solution techniques based on model-to-model transformations (e.g., to QNs or 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 [3, 7].

A DML instance (see Figure 1) 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*.

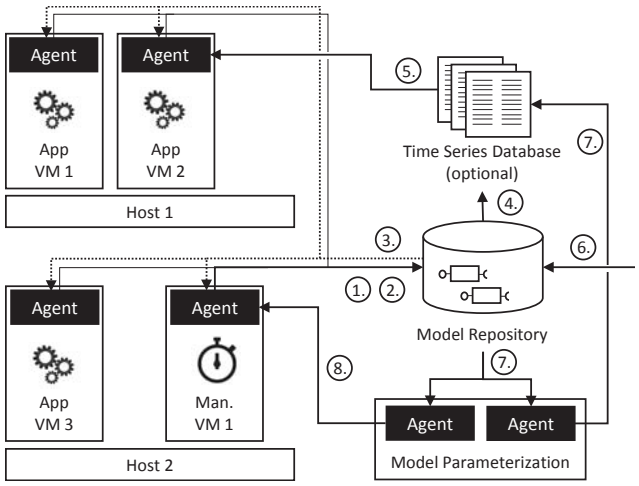

Figure 2: Reference architecture.

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 1). 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.

4. REFERENCE ARCHITECTURE

The complexity of today's virtualized environments requires a shift from monolithic model extraction solutions to a distributed, component-based one, in order to reduce the effort for tailoring the model extraction for certain technologies and to increase the reuse of functionality. Figure 2 shows the core components of our reference architecture, which are a set of agents, a central model repository, and optionally, one or multiple time series databases.

Agent: We consider three classes of agents: model skeleton, monitoring and model parameterization. A *model skele-*

ton consists of a partial DML model and a specification of the types of monitoring data that can be observed at the system. Model variables in the model skeleton (e.g., resource demands, branching probabilities) are marked as either *explicit* or *empirical*. Explicit model variables are assumed to have a fixed value (or a stochastic expression as supported by DML) that does not change continuously over time. In contrast, the values of empirical model variables are derived from monitoring data. *Monitoring* agents provide access to data sources containing the required monitoring data. *Model parameterization* agents determine the values of empirical model variables for a given time instant using the monitoring data. This time instant may also lie in the past (depending on the availability of historic monitoring data) or in the future (e.g., leveraging forecasting techniques).

Model repository: The model skeletons provided by the different agents are merged into a combined DML instance representing the complete system, which is stored in the model repository. This model is still unparameterized, i.e., all empirical model variables do not have values.

Time series databases: The time series databases are used to collect and store historic observations of different metrics (e.g., utilization, throughput and response times). The time series databases are optional, because the same information can be retrieved directly from monitoring agents. However, the availability of historic data may be limited directly at the monitoring agents. Time series database may be used to store historic data for a longer period or to improve the data access performance.

4.1 Assumptions

The architecture is based on the following assumptions:

- Model skeletons need to be composable. We assume that a model skeleton is always a valid, but not necessarily a complete model according to the DML metamodel. This ensures that the model skeletons can be merged automatically (see Section 4.4) at run-time.
- The model repository may be accessed by different agents concurrently. Therefore, the model repository provides transactional access to ensure atomicity for multi-step model updates. Furthermore, it requires locking capabilities to detect and prevent concurrent modifications of model elements.
- In order to be able to resolve connections between model skeletons, certain elements require unique identifiers (see Section 4.4).

4.2 Design Decisions

The design of agents for a specific system or technology requires a number of design decisions that need to be taken into account.

Functionality: In the simplest case, an agent just delivers a prepackaged model skeleton when the agent is started (e.g., capturing a-priori knowledge about an application). However, in many cases the model skeleton depends on the configuration of the operating system, middleware system (e.g., which components are deployed in a runtime container) or application (e.g., customizations in the application settings). An agent may use static and dynamic analyses to construct such a model skeleton at runtime.

Granularity: In theory, an agent may have different levels of granularity. For instance, an agent may be model skeleton and monitoring agent at the same time. While this increases the implementation complexity of the agent, it may be beneficial when integrating existing model extraction tools, such as [2] or [4], into the reference architecture. The existing tool may be reused as a whole only requiring a transformation from its output format to a model skeleton (based on DML). Furthermore, there are different agent roles (see Section 4.5) which may be split between agents.

Genericity: It is the agent designer's decision how generic an agent is designed (i.e., how much technology-specific knowledge is included in it). Model skeleton and monitoring agents may often be specifically designed for a certain technology (e.g., certain JEE application server product) in order to be able to fully exploit proprietary instrumentation and introspection capabilities. Furthermore, a deeper understanding of the underlying technology also may be required for mapping to concepts of DML (e.g., what is a component, or what is a container?). On the other hand, model parameterization agents will typically be generic as they directly work on the DML model and time series monitoring data.

Distribution: The distribution of the agents in Figure 2 is just exemplary, and not prescribed by the reference architecture. For instance, model skeleton and monitoring agents may not be required for each VM of an application, if an existing system or application monitoring solution is used (such as Dynatrace[1] or Kieker [16]) that provides all required information for creating the model skeleton in a central place.

Deployment: The agents may be deployed in the same VM as the application itself or in dedicated VMs. For monitoring and model skeleton agents, a deployment directly alongside the monitoring tool or the application itself may be beneficial (e.g., easier access to introspection interfaces or monitoring files). Model parameterization agents may depend on computationally expensive calculations (e.g., for resource demand estimation) and a deployment in a dedicated VM avoids negative impacts on the application performance.

Notification: Reconfigurations in the environment or changes in the workload may require updates to the performance model. The agents may either work in push or pull mode. In push mode, the agent exploit special notification mechanisms of the infrastructure or application software in order to be informed of changes. In pull mode, the agents check for changes in regular intervals.

4.3 Communication

The agents and the monitoring repository need to communicate with each other in order to build a complete model of the systems. Figure 2 gives an overview of the communication paths described in the following:

1. When a new agent is started, it automatically registers itself at the model repository. The address of the model repository needs to be configured in the agent. The model repository manages a list of agents in the environment. The same applies to time series databases.
2. The model skeleton agents write their current model skeleton to the model repository. The model repository automatically merges the skeleton into a single DML model. Monitoring agents also send information about the sensors in the environment. This step may

[1] http://www.dynatrace.com

be repeated if any changes in the environment are detected.

3. The agents may optionally register for notifications when the contents of the model repository changes. In this step, the agents are informed of the updated model.
4. The model repository may instruct the time series database to regularly collect and store metrics from the monitoring agents.
5. The time series database collects performance metrics in regular intervals from monitoring agents.
6. A user or another program may request a fully parameterized model for a defined instant in time from the model repository.
7. The model parameterization agents are triggered to provide updated values for the model variables (e.g., resource demands).
8. The model parameterization agent asks the time series database or a monitoring agent for the current monitoring data that is required to deteremine the values of a model variable.

4.4 Model Skeletons

A model skeleton represents the local view of an agent on the virtualized system. Different model skeleton agents may be responsible for different parts of the system. For instance, an agent at the virtualization layer can determine the physical hosts and the VMs running on each host, but cannot see what is running inside a VM. This information needs to be provided by other agents that have access to the applications inside the VM.

Meta-model: A model skeleton is described by a Meta Object Facility (MOF) compliant meta-model. Figure 3 gives an overview of this meta-model. A `ModelSkeleton`

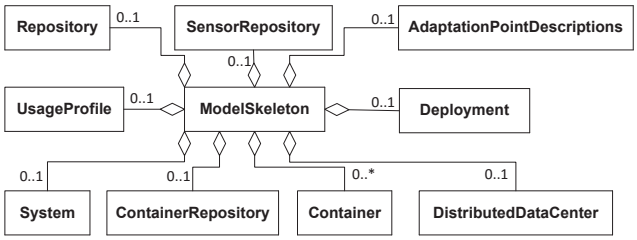

Figure 3: Model skeleton meta-model.

references six sub-models of the DML meta-model (see [3, 7]): `Repository`, `UsageProfile`, `System`, `ContainerRepository`, `DistributedDataCenter`, `Deployment`, and `AdaptationPointDescriptions`. These sub-models however contain only elements which are part of the local view of the agent, i.e., they are not a complete representation of the system. Therefore, all elements of a model skeleton are optional. The `Container` elements describes the resource layers within one or several VMs (e.g., middleware resources). The `SensorRepository` contains information about the sensors in the system. Figure 4 shows the corresponding meta-model (not part of DML).

The `SensorRepository` contains a list of `Sensor` definitions. Each `Sensor` represents one instrumentation point in the real system, where monitoring data is collected. A `Sensor` is defined by an `Agent`, an `Entity` (i.e., an arbi-

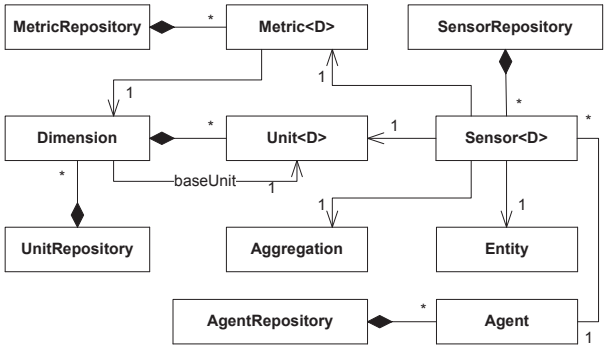

Figure 4: Sensor meta-model.

trary DML element in the other sub-models of the model skeleton), a `Metric` (e.g., response time, throughput, or utilization), a `Unit` (e.g., seconds), and an `Aggregation` (e.g., mean, minimum, or sum). `Sensor`, `Metric` and `Unit` are generic classes parameterized with a sub-class of `Dimension` (e.g., `Time`). The sub-classes are not depicted for reasons of conciseness. `MetricRepository` and `UnitRepository` are not part of the model skeleton. These are global registries for standard metrics and units.

Merging: In order to create a DML instance for the complete system, the model skeletons of different agents need to be merged. This is done automatically by copying the model skeletons one after the other to a target DML instance. Given that the model skeletons are created independently by different agents, we need to consider strategies for *matching* the same elements in different model skeletons and solving *conflicts* between model skeletons.

In order to prevent conflicts, we introduce an ownership model for the model skeletons. For certain meta-model elements, the internals of the element (i.e., attributes and containment references) can only be defined by a single agent (who is in the role of the owner). Other agents may also reference these model elements in their model skeletons (e.g., as the target of an assembly connector), however, they may not change the internals of these model elements. Thus, we can avoid non-resolvable conflicts between model skeletons. If two agents change the internals of the same model elements, it is detected by the model repository and reported to the agent as an error. Such errors should only occur if an agent's implementation is misbehaving, or the agents are configured incorrectly (e.g., two agents are monitoring the same entity in a system).

The only exception to this rule are component definitions in the repository. The component definitions are on a type level, i.e., they describe all possible service behaviors of a component. Given that we may observe different service behaviors for different instances of the same component, these behaviors need to merge into one behavior description. Conflicts may happen with fine-granular service behavior descriptions, which describe the intra-component control flow. In this case, conflicting paths can be resolved automatically by introducing additional branching actions describing the alternative behaviors.

The matching of elements in different model skeletons is based on their names. Agents need to derive names from readily available technical information (e.g., class names,

URLs, IPs). This names need to be unique within a certain namespace. For instance, components and interfaces need to be only unique within the same application, containers within the same data-center and system providing roles globally. It is the responsibility of the designer of an agent to decide how to derive such unique names from existing technical identifiers.

4.5 Agent Roles

An agent may have one or multiple roles in the model extraction. A role determines for what parts of a DML model an agent can take over ownership. In the following, we describe the possible agent roles.

Resource landscape: The agent is responsible to extract the containers (e.g., physical hosts, VM, EJB container) of a resource environment. This also includes the active and passive resources of the container. Multiple agents may be involved in the creation of a complete container hierarchy. Furthermore, the agent is also responsible for the adaptation points of the owned containers. The result is stored in the `DistributedDataCenter`, `ContainerRepository`, `Container` and `AdaptationPointDescriptions` elements of the model skeleton.

Components: An agent with this role extracts component types and their interfaces including signatures. For each component, the agent determines the interface providing and interface requiring roles. For each component service it derives service behavior descriptions. It is important to note, that the agent does not take ownership of the components created by it. The result is stored in the `Repository` element of the model skeleton.

Component assembly: An agent with this role takes ownership of a (sub)system or a composite component. It is responsible to determine the assembly contexts contained in the system or composite component as well as the assembly connectors, providing and requiring delegation connectors. Furthermore, it can define stochastic parameter dependencies on the input parameters. An agent owning a (sub)system also needs to determine the deployment of assembly contexts. The result is stored either in the `Repository` or in the `System/Deployment` elements of the model skeleton.

Usage scenario: An agent with this role takes ownership of a usage scenario. This includes the workload type as well as the usage scenario behavior (system call actions, branches, loops, etc.). The result is stored in the `UsageProfile` and `System` elements of the model skeleton.

Model variable: An agent with this role takes ownership of individual model variables in the model. A model variable may be the response-time behavior of a component (i.e., a description of the black-box response time distribution), a branching probability in a usage scenario behavior, the load intensity in a usage scenario, the branching probabilities and resource demands in a service behavior description or the influencing parameters in a parameter dependency.

5. IMPLEMENTATION

In this section, we describe a reference implementation of the model repository and agents for different technologies.

5.1 Model Repository

The implementation is based on MOF-based technologies. All meta-models are described using the Eclipse Modeling Framework (EMF)[2]. The model repository is implemented using the Eclipse Connected Data Objects (CDO) technology[3]. CDO is a distributed, shared data model based on EMF. It provides persistent storage and transactional access for EMF-based model instances. When applying a model skeleton to the model repository, the agent uses a single atomic transaction to execute all changes on the central DML instance. Thus, it is ensured that other agents do not see any invalid model states when applying a model skeleton. CDO also provides locking mechanisms to avoid inconsistencies due to concurrent modifications in the model repository. We use an optimistic locking scheme to avoid the overhead of explicit locking. If applying a model skeleton fails due to concurrent modifications, the transaction is repeated a configurable number of times.

CDO contains a distributed notification service. An agent can register itself to receive notifications if a certain model element is changed in the repository. The logic for applying and merging a model skeleton into the model repository is contained in a shared Java library and integrated into the agents. This library automatically checks that no elements owned by another agent are overwritten.

The current implementation assumes that all VMs in a data-center are connected to the same network, so that the IPs of a VM can be considered as being unique within the data-center. It is possible to use a dedicated network for the model extraction that is separated from the network(s) used for accessing the applications. Containers in the DML resource landscape model are then identified by their IP, so that different agents in the system can find the matching containers. An effective security isolation between applications of different tenants (e.g., in a public cloud system) is currently not implemented, although CDO provides mechanisms to implement access control schemes. This is left for future work.

CDO automatically tracks the revision history of a model. Thus, it is possible view the change-log of a model. On the one hand, this can be used for auditing purposes showing which agent changed what. On the other hand, it is possible to answer historic queries to the model repository (e.g., give me the performance model representing the system 3 days ago). In such cases, one would need to look up the historic performance model and trigger the model parameterization agents to determine the historic values for the model variables. Currently, our implementation only supports the retrieval of the latest performance model version.

5.2 Agents

In order to show that the described reference architecture can be realized in real systems, we have implemented several exemplary agents for different purposes. In the following, we shortly describe how these agents work.

VMware vSphere: The VMware vSphere server hypervisor provides a VCenter server virtual appliance to manage all VMs in a data center consisting of several physical hosts. The VCenter server manages an infrastructure repository that contains detailed information about the configuration of all hosts and VMs. The agent can access this information through a web services interface [17] provided by VCenter server. As a resource landscape agent, it extracts all containers up to the level of VMs. Additionally, it also has

[2]https://eclipse.org/modeling/emf/
[3]https://eclipse.org/cdo/

the role of a monitoring agent providing resource utilization data for hosts and VMs. The agent registers itself at the VCenter server to be notified of any changes in the configuration using the PropertyCollector managed object [17]. Any changes to the hypervisor configuration (e.g., starting or stopping VMs, live migrations) are detected and the resource landscape model is updated accordingly.

WildFly[4] (former JBoss) is a JEE 7 application server. It is based on an OSGi framework and thus provides deep extension mechanisms. Our agent is implemented as a Wildfly extension and runs directly in the application server. Thus, it has direct access to all services of the application server supporting deep introspection. The agent registers itself as a custom deployment unit processor by implementing the `org.jboss.as.server.deployment.DeploymentUnitProcessor` interface. It is called whenever a new application archive is deployed on the application server. The current implementation, automatically registers a set of interceptors at incoming and outgoing calls of components (Enterprise Java Beans, web services, servlets) to monitor the control flow at run-time. The WildFly agent is a model skeleton agent with the components and component assembly roles as well as a monitoring agent providing throughput data for component services.

LibReDE is a library for resource demand estimation [13]. Based on this library we have implemented a model parameterization agent that determines values for the resource demands based on monitoring data. The agent expects the unparameterized DML instance and searches for resource demands in the model marked as empirically. With the information in the sensor repository, it then automatically decides which approaches to resource demand estimation are applicable. After resource demand estimation, the results are stored in the model repository.

Generic template: This is a generic implementation of a model skeleton agent that just delivers a prepackaged model skeleton from the file system of a VM. The model skeleton needs to be created manually. For example, we currently use this agent to deliver the usage profile and a black-box description of the database as long as we do not have specialized agents for these parts of the system.

6. CONCLUSIONS

In this paper, we have presented a reference architecture for online model extraction in virtualized environments. The reference architecture enables the design of technology-specific agents while increasing the reuse of generic model extraction functionality. Furthermore, it enables the consolidation of a complete architecture-level performance model from heterogeneous data sources in data center.

As future work, we plan to work on the following topics: (a) Integration of existing model extraction approaches (such as [2, 4]) into our reference architecture. (b) Integration of more model parameterization tools (such as [5]) into the reference architecture. (c) Support for security isolation between applications of different tenants in a data center.

7. ACKNOWLEDGMENTS

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

[4]http://wildfly.org/

8. REFERENCES

[1] M. Awad and D. A. Menascé. Dynamic derivation of analytical performance models in autonomic computing environments. In *Proceedings of the 2014 Computer Measurement Group Conference*, 2014.

[2] F. Brosig, N. Huber, and S. Kounev. Automated extraction of architecture-level performance models of distributed component-based systems. In *26th IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE 2011)*, pages 183–192, 2011.

[3] F. Brosig, N. Huber, and S. Kounev. Architecture-level software performance abstractions for online performance prediction. *Sci. Comput. Program.*, 90:71–92, 2014.

[4] A. Brunnert, C. Vögele, and H. Krcmar. Automatic performance model generation for java enterprise edition (EE) applications. In *Computer Performance Engineering - 10th European Workshop, EPEW 2013*, pages 74–88, 2013.

[5] N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn. Self-adaptive workload classification and forecasting for proactive resource provisioning. *Concurrency and Computation: Practice and Experience*, 26(12):2053–2078, 2014.

[6] C. E. Hrischuk, C. M. Woodside, J. A. Rolia, and R. Iversen. Trace-based load characterization for gernerating performance software models. *IEEE Trans. Software Eng.*, 25(1):122–135, 1999.

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

[8] A. Koziolek, D. Ardagna, and R. Mirandola. Hybrid multi-attribute QoS optimization in component based software systems. *Journal of Systems and Software*, 86(10):2542 – 2558, 2013.

[9] K. Krogmann. *Reconstruction of software component architectures and behaviour models using static and dynamic analysis*. PhD thesis, Karlsruhe Institute of Technology, 2010.

[10] D. Menascé, V. Almeida, and L. Dowdy. *Capacity Planning and Performance Modeling: from mainframes to client-server systems*. Prentice Hall, 1994.

[11] D. A. Menascé, V. Almeida, R. C. Fonseca, and M. A. Mendes. A methodology for workload characterization of e-commerce sites. In *EC*, pages 119–128, 1999.

[12] S. Spinner, G. Casale, F. Brosig, and S. Kounev. Evaluating Approaches to Resource Demand Estimation. *Performance Evaluation*, 92:51 – 71, October 2015.

[13] S. Spinner, G. Casale, X. Zhu, and S. Kounev. LibReDE: A Library for Resource Demand Estimation. In *5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014)*, pages 227–228, March 2014.

[14] S. Spinner, S. Kounev, X. Zhu, and M. Uysal. Towards Online Performance Model Extraction in Virtualized Environments. In *8th Workshop on Models @ Run.time (MRT 2013)*, pages 89–95, Sept 2013.

[15] A. van Hoorn, C. Vögele, E. Schulz, W. Hasselbring, and H. Krcmar. Automatic extraction of probabilistic workload specifications for load testing session-based application systems. In *8th Intl. Conf. on Performance Evaluation Methodologies and Tools*, 2014.

[16] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*, pages 247–248, April 2012.

[17] VMware, Inc. Vmware vsphere web services sdk documentation. https://www.vmware.com/support/developer/vc-sdk/. Accessed 20.11.2015.