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 previously in our vision described 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
architecture-level performance models [9, 2, 4]. Hirschuk 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 Brummert 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 the extraction process. Different statistical techniques for estimation of resource demands based on monitoring data have been proposed (see the survey and experimental comparison by Späni 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 OPNs). 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 an 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 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 skel-
ton consists of a partial DML model and a specification of
the types of monitoring data that can be observed at the sys-
tem. Model variables in the model skeleton (e.g., resource
demands, branching probabilities) are marked as either ex-

demands, branching probabilities) are marked as either ex-

plicit 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 monitor-
ing data. This time instant may also lie in the past (depend-
ing 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 di-
rectly at the monitoring agents. Time series database may
be used to store historic data for a longer period or to im-
prove 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 neces-
sarily a complete model according to the DML meta-
model. 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 order to be able to resolve connections between
model skeletons, certain elements require unique iden-
tifiers (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 deliv-
ers 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 con-
tainer) 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
skleton and monitoring agent at the same time. While
this increases the implementation complexity of the agent,
it may be beneficial when integrating existing model extrac-
tion 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 skele-
ton (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 knowl-
edge 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 intro-
spection capabilities. Furthermore, a deeper understanding
of the underlying technology also may be required for map-

ting to concepts of DML (e.g., what is a component, or what
is a container?). On the other hand, model parameteriza-
tion 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 moni-

toring 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 dedi-
cated VM avoids negative impacts on the application per-
formance.

Notification: Reconfigurations in the environment or
changes in the workload may require updates to the perform-
ance 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 com-
municate with each other in order to build a complete model
of the systems. Figure 2 gives an overview of the communi-
cation paths described in the following:

1. When a new agent is started, it automatically regis-
ters 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 reposi-
tory automatically merges the skeleton into a single
DML model. Monitoring agents also send information
about the sensors in the environment. This step may
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 determine 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 references six sub-models of the DML meta-model (see [3, 7]): System, Repository, UsageProfile, 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 describe 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 metamodel (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 arbitrary 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,
5.1 Model Repository

The implementation is based on MOF-based technologies.

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.
tainers up to the level of VMs. Additionally, it also has 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 a 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.

8. REFERENCES


4http://wildfly.org/