2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures

Considering Transient Effects of Self-Adaptations in Model-Driven Performance Analyses

Christian Stier
FZI Research Center for Information Technology
Karlsruhe, Germany
stier@fzi.de

Anne Koziolek
Karlsruhe Institute of Technology
Karlsruhe, Germany
koziolek@kit.edu

***Abstract*—Model-driven performance engineering allows software architects to reason on performance characteristics of a software system in early design phases. In recent years, model-driven analysis techniques have been developed to evaluate performance characteristics of self-adaptive software systems. These techniques aim to reason on the ability of a self-adaptive software system to fulfill performance requirements in *transient phases*. A transient phase is the interval in which the behavior of the system changes, e.g., due to a burst in user requests. However, the effectiveness and efficiency with which a system is able to adapt depends not only on the time when it triggers *adaptation actions* but also on the time at which they are completed. Executing an adaptation action can cause additional stress on the adapted system. This can further impede the performance of the system in the transient phase. Model-driven analyses of self-adaptive software do not consider these *transient effects*. This paper outlines an approach for evaluating transient effects in model-driven analyses of self-adaptive software systems. The evaluation applied our approach to a horizontally scaling media hosting application in three experiments. By considering the delay in booting new Virtual Machines (VMs), we were able to improve the accuracy of predicted response times. The second and third experiment demonstrated that the increased accuracy enables an early detection and resolution of design deficiencies of self-adaptive software systems.**

I. Introduction

Modern software systems face the need to meet quality requirements under changing user load. Cloud computing enables software architects to design their applications to dynamically provision resources based on the current and expected user load. Infrastructure as a Service (IaaS) clouds offer this flexibility in the shape of dynamically provisioned VM instances. Deciding *where* and *when* to adapt have implications that go beyond implementation or operational aspects. The architecture of a software system needs to be designed to enable the *scaling* of subsystems based on the user load. Model-driven evaluation techniques for software systems [1] allow software architects to reason on the effect of design decisions on the quality of the system. As these techniques operate on abstractions of the developed system, they can be applied in early design stages in which no implementation is available. They can also be used to evaluate the impact of design decisions before they are implemented. Traditionally, model-driven analysis techniques [1] such as Palladio [2] focus on analyzing systems with static structure, usage and deployment. Approaches like SimuLizar [3] and SLAstic.SIM [4] extend these techniques to the architectural design of self-adaptive software systems.

Self-adaptive software systems adapt their structure, deployment and configuration to efficiently and effectively fulfill Quality of Service (QoS) requirements under changing user load. Adapting the configuration of a software system takes time and consumes resources. If a horizontally scaling application using IaaS does not provision additional VMs in time to deal with higher user demand, the system's performance drops and performance requirements might not be met in the transient phase. Conversely, provisioning resources too early results in avoidable cost. *Transient effects* refer to the performance impact caused by changes in the user load and the adaptations made to address them. An adaptation only improves the QoS of a system if the benefit gained from it outweighs its temporary negative impact on performance. It is important that a self-adaptive system is designed so that the adaptations it makes do not worsen QoS by using already congested resources. Reasoning on the behavior of self-adaptive software systems in transient phases needs to consider provisioning times and resource overhead caused by adaptation actions.

Model-driven performance analyses for self-adaptive software systems such as SimuLizar [3] and SLAstic.SIM [4] currently neglect transient effects. This leads to overly optimistic predictions as adaptations are assumed to induce no resource demand and to complete instantaneously. Existing performance models that consider transient effects only incorporate the effects for a specific system [5] or model them on an abstraction level that is unsuited for architecture-level analyses [6], [7].

We propose a methodology for defining and evaluating the impact of transient effects on the performance of self-adaptive software systems. Our approach supports the systematic modeling of *adaptation actions* including their transient effects expressed as resource demands. An example for an adaptation action is *scale-out*. In IaaS systems, scale-out distributes load to additional replicas of a sub-system deployed on new VMs.

We model adaptation actions to depend only on the *types* and *roles* of entities they adapt. This enables a reuse of adaptation action specifications including a reusable transient effect specification. We model the transient effect of adaptation actions in parametric performance models. The parametric performance model defines the performance effect of adaptation actions independent of the architecture model of a specific system. Using parametrized performance models allows us to specify both the effect adaptation actions have on performance in transient phases, as well as the dependency of adaptation actions on system load. Examples of parameters that influence the time to complete a scale-out are the chosen image and the

978-1-5090-2567-1/16 $31.00 © 2016 IEEE
DOI 10.1109/QoSA.2016.14

deployment environment, as shown by Mao et al. [8].

The contributions of this paper are as follows. First, it presents an approach for coupling the specification of self-adaptation mechanisms with (i) a specification of their effect on system performance, and (ii) the effect of system performance on the execution time of self-adaptation mechanisms. Second, it outlines an analysis methodology that allows to consider the performance effect of executing adaptations in model-driven performance analyses of self-adaptive systems.

To evaluate the benefit gained from considering transient effects in software performance analyses we extended SimuLizar's [3] model-driven analysis of self adaptive software systems with our approach. The evaluation compared the baseline SimuLizar analysis against the analysis extended by our approach with measurements conducted for a reference implementation of the system under evaluation. We conducted three experiments using the web-based media hosting application Media Store [9]. The application automatically scaled the number of VMs used to serve user requests depending on measured response times. The experiments compared the predicted with the observed system performance and behavior. The first two experiments showed that the consideration of transient effects of horizontal scaling reduced the mean error of the predicted average response times in transient phases. Furthermore, the experiments showed that our approach improved the accuracy of other quality metrics such as the maximum response time and degree of horizontal scaling. The second and third experiment illustrated that the increased accuracy in performance predictions enables software architects to detect and address deficiencies in the design of self-adaptive software systems.

This paper is structured as follows. Section II outlines foundations. Section III introduces our model for describing transient effects of adaptation actions. Section IV formally describes the concepts of the model. Section V discusses how the model is interpreted in a model-driven performance analysis. Section VI presents the evaluation results. Section VII discusses related work. Section VIII concludes.

II. Foundations: Model-driven Performance Analyses of Self-adaptive Software Systems

Traditional model-driven performance analyses [1] reason on the performance of static software systems. Model-driven performance analyses of self-adaptive software systems extend these techniques to the domain of self-adaptive systems. These analyses consider the effect of adaptations performed by self-adaptation mechanisms to deal with changes in user load. This enables reasoning on the behavior of self-adaptive systems in transient phases.

SimuLizar by Becker et al. [3] is an architecture-level performance analysis methodology for self-adaptive software systems. SimuLizar supports the analysis of self-adaptive software systems that conform to the MAPE-K control feedback loop [10]. SimuLizar can be used to evaluate QoS of a self-adaptive system for a set of workload distributions or workload traces. SimuLizar uses an extended Palladio Component Model (PCM) [2] to describe both the design-time and runtime architecture of a self-adaptive software system.

The design-time architecture model defines the initial structure of the system before it is adapted. The runtime architecture model is changed as part of the analysis whenever a self-adaptation mechanism adapts the system under analysis. Palladio's design-time architecture model describes component specification, assembly and deployment. SimuLizar extends these PCM viewpoints with a *monitoring specification viewpoint* and *state transition* viewpoint. The monitoring specification viewpoint describes which measurements should be collected from the system under analysis and passed to self-adaptation rules. The state transition viewpoint allows to specify self-adaptation rules using model transformations or graph-based pattern rules. Self-adaptation rules use the PCM runtime architecture model and measurements specified by the monitoring specification viewpoint. If the runtime architecture model and measurements meet the adaptation conditions of an adaptation rule, the rule triggers a system reconfiguration by transforming the runtime architecture model. SimuLizar adjusts the system state of the analyzed system to reflect the changes made to the runtime architecture model. Currently, SimuLizar's implementation supports the specification of adaptation rules in QVT Operational (QVTo) [11], Story Diagrams [12] and Henshin [13].

SimuLizar considers the effect of the self-adaptation rules as part of of its simulation-based performance analyses. Software architects can use SimuLizar to evaluate whether a self-adaptive system fulfills requirements in transient phases [3]. However, Becker et al. make a set of limiting assumptions in the system model of SimuLizar. The authors assume that reconfiguration actions execute instantaneously and cause no further resource demand. Consequently, SimuLizar only allows to identify whether a self-adaptive software system manages to fulfill requirements in transient phases if the considered adaptations require a negligible amount of time to execute and do not impact system load. This neglects the transient effect of self-adaptation actions such as VM migration [14] and VM instantiation [8]. Furthermore, SimuLizar does not consider the resource demand caused by evaluating a self-adaptation mechanism.

III. Modeling Transient Effects

This section outlines our approach for modeling the transient effects of adaptation actions. Adaptation actions do not complete instantaneously. When a scale-out action replicates a sub-system on a new VM, the newly instantiated sub-system is not available immediately. The VM first has to be booted, and the sub-system has to be deployed and launched on it. Existing model-driven performance prediction approaches for self-adaptive software systems do not consider or model these transient effects.

Our *Self-Adaptation Action Model* links the effect of adaptation actions on the configuration of the system with the performance effect that is caused by executing each action. A *Self-Adaptation Action* is a set of atomically executable middleware operations. Self-adaptation frameworks such as Stitch [15] or Descartes [16] combine individual actions to complex adaptation mechanisms. As our Adaptation Action Model couples each action with a specification of its performance effect, we can derive the performance effects of self-adaptation mechanisms that consist of Adaptation Actions.

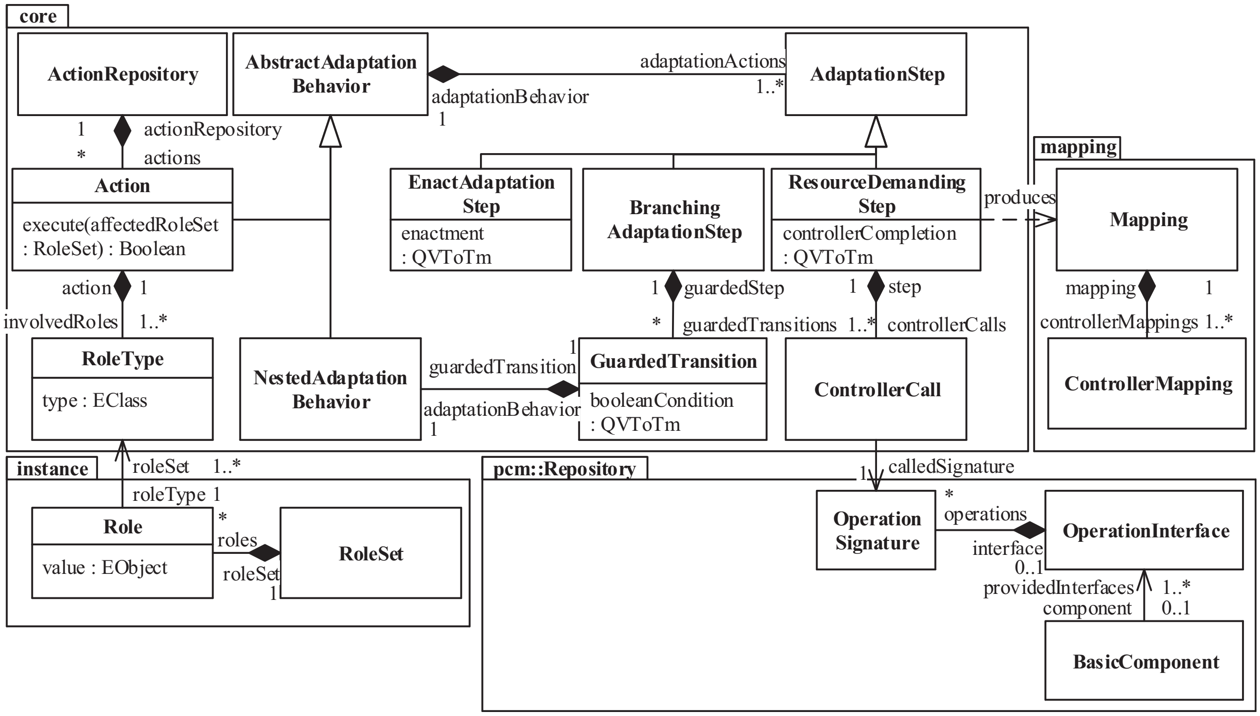

Fig. 1: Overview of the Self-Adaptation Action Meta-Model

Figure 1 gives an overview of the Self-Adaptation Action Model's meta-model. The model can be used to describe *types of Self-Adaptation Actions* for use in a model-driven performance analysis. Its pragmatism is to couple the specification of the *effect* of Adaptation Actions on system configuration with their performance effects. Individual actions specified using our model are parametrized. Once an action has been defined and put in an *ActionRepository*, it can be reused to consider transient effects in different architecture models of self-adaptive software systems.

To illustrate our model, let us consider horizontal scaling of a software system, which aims to adapt the number of provisioned resources to the user load. A self-adaptation mechanism may implement the provisioning and use of additional resources as a scale-out action. Figure 2 shows a scale-out action in an IaaS cloud environment described using our model. *ScaleOut* consists of a set of *AdaptationSteps* and *RoleTypes*. A RoleType represents a parameter that an action depends upon. In our example, *ScaleOut* replicates an assembly of components and distributes load to the replica. The RoleType *ScaledComponentInstance* specifies the replicated component as an input parameter.

Our model describes an adaptation action as an ordered set of *AdaptationSteps*. A *BranchingAdaptationStep* describes conditions that must be met before an adaptation. In our scale-out example, *CheckAndExecute* subsumes the conditions (*CheckPreconditions*) for scaling out. An example for a condition is that the upper bound of available VMs has not been reached. *ExecuteScaleOut* encompasses the steps that are to be executed if the conditions are met.

The adaptations' effect on system performance is expressed by a *ResourceDemandingStep*. In our example, *InstantiateVm* describes the transient effects induced by instantiating a VM. It consists of the *Instantiate* call which describes the performance effect of instantiating a new VM. *Instantiate* references the *Adaptation Performance Model* which describes the resource demands caused by executing an Adaptation Action. Application Performance Models consist of *Controller* components. This paper uses PCM components [2] to model Controller components. The *ComponentInstantiationController* in the Adaptation Performance Model links to the VM host (*ComponentInstantiationLocation*) since component instantiation can affect the performance on both the server hosting the cloud controller and the host on which the controller deploys the VM. The *controllerCompletion* QVTo model transformation of a *ResourceDemandingStep* describes how to add the Adaptation Performance Model to the performance model of a system based on the parameters of the *AdaptationStep*. These parameters are passed to the action as a set of *Roles* which conform to the RoleTypes of the action.

The adaptations' effect on system configuration is expressed by an *EnactAdaptationStep*. Its *enactment* QVTo model transformation describes the effect of the adaptation on the runtime architecture model. In our example, *ProvisionComponents* describes the effect a scale-out action has on the assembly and deployment of the components. At analysis time, it adds the instantiated VM to the PCM runtime architecture model, provisions the scaled components and adds the components to the load balancer that distributes load among the components.

IV. Formalization

Section III illustrated our model for describing transient effects in self-adaptive systems. This section formally describes

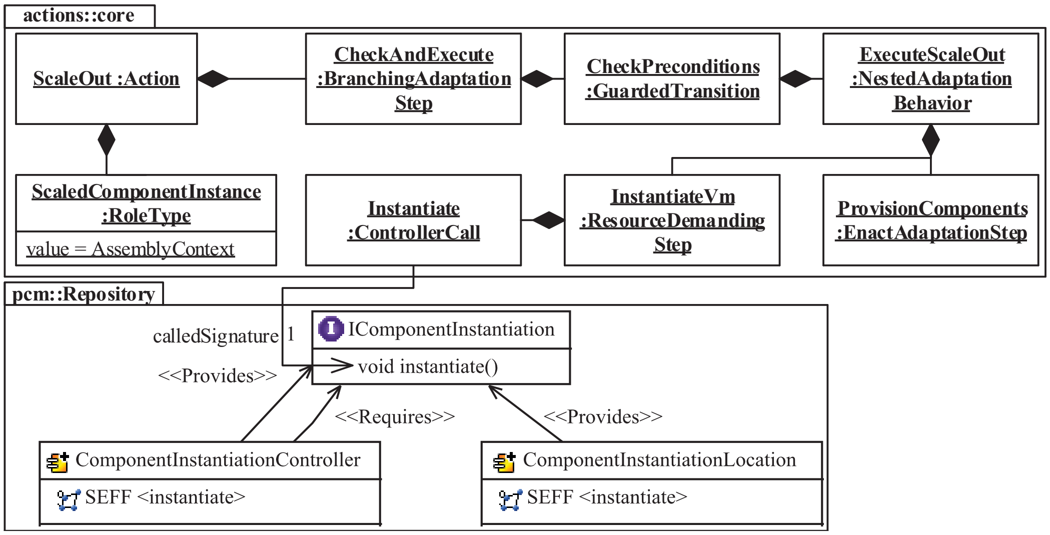
actions::core
ScaleOut :Action
CheckAndExecute :BranchingAdaptation Step
CheckPreconditions :GuardedTransition
ExecuteScaleOut :NestedAdaptation Behavior
ScaledComponentInstance :RoleType
value = AssemblyContext
Instantiate :ControllerCall
InstantiateVm :ResourceDemanding Step
ProvisionComponents :EnactAdaptationStep
pcm::Repository
calledSignature
1
IComponentInstantiation
void instantiate()
<<Provides>>
<<Requires>>
<<Provides>>
ComponentInstantiationController
SEFF <instantiate>
ComponentInstantiationLocation
SEFF <instantiate>

Fig. 2: Self-Adaptation Action Model of Scale-Out Action

the fundamental concepts of our model which are independent of a specific performance model or architectural modeling language. The formalization presented here builds upon the self-adaptive system model as defined by Becker et al. [3][1]:

Definition 1. Self-Adaptive System Model (based on [3])
A self-adaptive system model is a tuple (S, E, σ), where

- S is the domain of all system states,
- E is the domain of monitored environment states,
- σ is the set of self-adaptation rules $\{\sigma_1, \ldots, \sigma_l\}$.

A system state $s \in S$ includes all aspects of a running system's state relevant for self-adaptation, ranging from the current component allocations to servers to the state of active requests, utilization of resources, and performance indices.

Additionally, we define M_l as the domain of all runtime architecture models with respect to an architecture description language l. $m_s \in M_l$ is an abstraction of the overall system state s that considers only architectural elements expressible in the used language l. For example, in PCM this view on the system state is concerned with the deployment of components to servers, component selection and configuration, etc. To represent the runtime management of a self-adaptive system, we define the function $\chi : S \times M \times \Phi \rightarrow S \times M \times \Phi$ as an idempotent mapping that ensures consistency between $m \in M$ and $s \in S$ and applies the identity function to an input $\phi \in \Phi$.

Our Adaptation Action Model decomposes adaptation rules σ into sequences of conditionally executed, parametrized Adaptation Actions, which again are decomposed into sequences of Adaptation Steps. For our definitions, let the operator $\hat{\circ}$ denote the partial composition of two functions $f_2 \hat{\circ} f_1$ that preserves all parameters except the parameters affected by f_1. Let $f_1, f_2 : \Gamma_1 \times \ldots \times \Gamma_j \times \Omega_1 \times \ldots \times \Omega_t \rightarrow \Gamma_1 \times \ldots \times \Gamma_j$, then $f_2 \hat{\circ} f_1(\gamma_1, \ldots, \gamma_j, \omega_1, \ldots, \omega_t) = f_2(f_1(\gamma_1, \ldots, \gamma_j, \omega_1, \ldots, \omega_t), \omega_1, \ldots, \omega_t)$, where $(\gamma_1, \ldots, \gamma_j) \in \Gamma_1 \times \ldots \times \Gamma_j$ and $(\omega_1, \ldots, \omega_t) \in \Omega_1 \times \ldots \times \Omega_t$.

$\phi = \{\phi_1, \ldots, \phi_m\} \in \Phi$ are a set of adaptation parameters, where Φ is the domain of Adaptation Action parameters.

Definition 2. Adaptation Action
An Adaptation Action is a functional composition $a(s, m, \phi) = \mu_1 \hat{\circ} \ldots \hat{\circ} \mu_n(s, m, \phi)$ of Adaptation Steps, where $s \in S$ and $m \in M_l$. Here, $\mu_i = p_i \circ \psi_i \circ \chi$ with $1 \leq i \leq n$ is the composition of an Adaptation Step $p_i \in P$, the selection function ψ_i and χ. ψ_i selects a fitting subset of $(s, m, \phi) \in S \times M \times \Phi$ depending on the type of p_i.

Definition 3. Adaptation Step
An Adaptation Step $p_i \in P$ describes an individual operation that is executed as part of an adaptation action. $P = P_{br} \cup P_{rd} \cup P_{\sigma}$ combines the domains of the different types of Adaptation Steps to couple the description of conditions (P_{br}) and reconfigurations (P_{σ}) with their performance effect (P_{rd}).

Definition 4. Branching Adaptation Step
A Branching Adaptation Step is a function $p_{br} : S \times M \times \Phi \rightarrow S \times M \times \Phi$

$$p_{br}(s, m, \phi) = \begin{cases} \mu_m^1 \hat{\circ} \ldots \hat{\circ} \mu_1^1(s, m, \phi) & \text{if } c_1(m, \phi) = true, \\ \ldots & \\ \mu_l^n \hat{\circ} \ldots \hat{\circ} \mu_1^n(s, m, \phi) & \text{if } c_n(m, \phi) = true, \\ (s, m, \phi) & else. \end{cases}$$

c_j is a function $c_j : M_l \times \Phi \rightarrow \{\text{true}, \text{false}\}$ that evaluates whether the runtime architecture model $m \in M_l$ and a set of passed parameters $\phi \in \Phi$ meet specific adaptation preconditions.

Definition 5. Resource-Demanding Adaptation Step
A Resource-Demanding Adaptation Step $p_{rd} \in P_{rd}$ is a function $p_{rd} : S \times \Phi \rightarrow S$.

p_{rd} introduces a transient effect of an Action in the system state. p_{rd} changes the current system state $s \in S$ by including the resource demands caused by executing an Adaptation Action, resulting in a new system state $s' \in S$.

VM migration is an example of an Adaptation Action for

[1] We dropped elements of the definition that are not relevant for this paper.

which the system state s and parameters ϕ affect the transient effect caused by executing it. The migrated VM and migration target are examples for ϕ. The transient effects of a migration depend on the system state in the shape of VM size and the rate with which the VM reads and writes data [14]. The additional network demand caused by VM migration takes up bandwidth which cannot be used by other services. If the network is overloaded in s, migration takes longer to execute.

Definition 6. Enact Adaptation Step
An Enact Adaptation Step $p_\sigma \in P_\sigma$ is a function $p_\sigma : M_l \times \Phi \rightarrow M_l$.

Unlike a Resource-Demanding Adaptation Step p_{rd}, an Enact Adaptation Step p_σ does not directly modify s. Each p_σ describes the effect of an action on the system's runtime architecture $m \in M_l$. p_σ transforms m to $m' \in M$ so that m' reflects the outcome of an adaptation operation.

Definition 7. Self-Adaptation Rules and Adaptation Actions
A self-adaptation rule $\sigma_t : S \times M \times E \rightarrow S \times M \times \Phi \in \sigma$ is defined as

$$\sigma(s,m,e) = \begin{cases} \chi \hat{o} k_v \hat{o} \ldots \hat{o} k_g \hat{o} \ldots \hat{o} k_1(s,m,e), & \text{if } c(m,e) = \text{true} \\ s, & \text{if } c(m,e) = \text{false} \end{cases}$$

- $c : M_l \times E \rightarrow \{\text{true}, \text{false}\}$ is the condition of the rule.
- $k_g(s,m,e) = a(s,m,\xi_g(m,e))$ is a parametrized call of an Adaptation Action. The function $\xi_g : M_l \times E \rightarrow \Phi$ projects the runtime architecture and environment state to appropriate Adaptation Action parameters $\phi \in \Phi$.

V. Analyzing Transient Phases in Self-Adaptive Software Systems

This section outlines an analysis methodology for considering transient effects in the performance analysis of self-adaptive software systems. It leverages the Self Adaptation Action meta-model shown in Figure 1 as a performance effect specification of Self-Adaptation Actions. This performance effect specification is used to reason on the transient effect of adaptations triggered by self-adaptation mechanisms.

The analysis methodology outlined in this section is compatible with model-driven performance analyses that use a system state model that can be modified to reflect the effect of self-adaptations. Simulation-based performance analyses like SimuLizar [3] or D-KLAPER [17] meet this requirement. Analytical methods for performance models, e.g. based on Layered Queueing Networks (LQNs) [18], do not support the analysis of transient phases in self-adaptive software systems. Hence, they also can not be extended to support the analysis of transient effects. Simulation-based performance analyses can use our analysis by calling the *execute* operations of Adaptation Actions specified using our model.

When a self-adaptation rule executes an Adaptation Action, our *Action Model Interpreter* enacts the execution of its Adaptation Steps. The Action Model Interpreter traverses the Action Model and executes the steps of an Action. Each call of execute of an Action is parametrized by a set of values that determine where, when and how it executes. The parameters passed in this *affectedRoleSet* instantiate the parameter types specified by the *involvedRoles* of the Action.

The Action Model Interpreter evaluates a *BranchingAdaptationStep* by evaluating the *booleanConditions* of each of its *guardedTransitions*. *booleanCondition* realizes the Boolean function c_j. Each GuardedTransition models a branch of the step. The interpreter executes the steps in the first branch j of p_{br} whose condition c_j evaluates to true.

The interpreter executes the *ResourceDemandingStep* in two phases. In the first phase, the interpreter calculates the union of runtime architecture model with the Adaption Performance Model. The *controllerCalls* of the *ResourceDemandingStep* reference the service signatures in the Adaptation Performance Model that are invoked to induce the performance effect.

The *RoleSet* passed to the Action via an *execute* call corresponds with the parameters ϕ. The interpreter adds the Adaptation Performance Model to the architecture runtime model $m \in M_l$ in a model transformation. We opted to let the interpreter add the Adaptation Performance Model to the runtime model m once a ResourceDemandingStep is called and not statically prior to the overall analysis due to the following reasons. To add the model to m prior to the analysis would require that the self-adaptive system's configuration space is predetermined, meaning that all possible states of the system can be preempted in a finite set. This is, however, not possible if the self-adaptive system's configuration space depends upon unbounded properties of the system environment. Even if it is possible to enumerate all potential states of a self-adaptive system, it may not be desirable to explicitly represent all states as even a few alternative adaptations result in state space explosion when considering potential combinations formed from the alternatives.

In the second phase the interpreter introduces the *resource demands* caused by the adaptation into the model-driven performance analysis. These resource demands are modeled in the Adaptation Performance Model. They may encompass CPU operations that need to be processed or a time frame that needs to pass before a step in an adaptation action completes. This resource demand needs to be explicitly considered in the model-driven performance analysis. In the case of the simulative analysis SimuLizar by Becker et al. [3] our interpreter starts an additional user that submits a user request for each *ControllerCall*. Each ControllerCall is parametrized by a subset of parameters from ϕ. Instead of enacting the adaptation caused by executing an action right away, the interpreter delays its execution until the demand caused by the ResourceDemandingStep has been processed. Only once all user requests have been completed the interpreter continues executing the next step.

The interpreter executes an *EnactAdaptationStep* by applying the model transformation *enactmentTm* to the model that represents the system state $s \in S$. *enactmentTm* implements p_σ.

VI. Evaluation

We investigated the following evaluation questions:

- **Q1:** Does our approach predict the effect of transient effects on an architectural level more accurately than existing approaches?
- **Q2:** Does our approach enable the (a) detection and (b) solution of design deficiencies of self-adaptive software systems?

To evaluate Q1, we compared the Response Time (RT) prediction error of the baseline SimuLizar [3] with the prediction error of a SimuLizar implementation we had extended with our approach. To evaluate Q2 we applied SimuLizar extended with our approach to detect and improve a design deficiency of a self-adaptive software system.

A. *Evaluated Application: Media Store*

We conducted the evaluation using an extended version of the *Media Store* reference implementation [9] of a lightweight media hosting service. Media Store has been used to validate the applicability of Palladio for design-time performance predictions [19]. It has also been used to evaluate the accuracy of Palladio's architecture-level performance [2] and energy efficiency [20] analyses.

Media Store users can choose to re-encode any music file they download from the web service. Reencoding is the most computationally intensive service offered by Media Store. We thus decided to extend Media Store's architecture to support horizontal scaling of its Reencoding subsystem. We encapsulated this subsystem in a separately deployable VM. A load balancer distributes incoming requests among all currently available Reencoding subsystems. We deployed all other Media Store components in the same VM.

We added a rule-based *Horizontal Scaler* component to the system to deal with changes in the user load by performing scale-out actions. Our intended goal was to maintain the requirement that the average Reencoding service response time over an interval of five minutes does not surpass two minutes. This requirement must be met for interarrival times between Reencoding requests that are as low as ten seconds. In order to maintain the requirement we designed the following set of conditions for a horizontal scaling rule:

1) The average response time of the Reencoding over the last five minutes is higher than twice the average response time.
2) No scale-out action has been started in the last five minutes.
3) No previous scale-out action is still being executed.
4) No more than *n* Reencoding VMs are used.

If all of the conditions are met, the Horizontal Scaler issues a scale-out action. The scale-out action launches an additional Reencoding VM and adds it to the load balancer distributing the requests as soon as it becomes available.

B. *Evaluation Setup*

We deployed the extended Media Store on a private IaaS OpenStack cloud. The OpenStack cloud used in the experiments was set up to deploy the VMs on a Dell PowerEdge R815 equipped with four Opteron 6174 CPUs running a Xen hypervisor. All Media Store components were deployed on CentOS 6.6 VMs. All components except for the Reencoder instances were deployed on a two core VM with 4 GB RAM. Reencoder instances were deployed on single core VM with 2 GB RAM. We used a separate physical machine connected via 1 GBit LAN running Windows 7 with an i7-2620M and 8 GB RAM to execute the JMeter 2.11 load driver. We ran a warmup load with an interarrival rate of 29 seconds and length of 10 minutes before conducting each of the 10 measurement runs per experiment scenario. For the performance simulations, we performed 100 simulation runs per scenario and simulator. The results, models and implementation used in the evaluation are available via[2].

TABLE I: Response Time Prediction Error per Workload Interval from the Comparison of 10 Measurement Runs and 100 Simulation Runs (Experiment A)

Interval	Median Error		Mean Error		Estd. Std. Dev.	
	Baseline	Extended	Baseline	Extended	Baseline	Extended
1	2.3%	2.3%	2.7%	2.7%	0.8%	0.8%
2	14.6%	5.4%	15.5%	6.2%	9.8%	4.5%
3	36.1%	33.7%	37.3%	42.8%	24.5%	38.8%
Total	22.5%	16.5%	22.7%	20.1%	14.1%	16.2%

C. *Accuracy of Predictions in Transient Phases*

This section investigates whether our approach is suited to accurately predict the performance of self-adaptive software systems in transient phases (Q1).

We evaluated the accuracy of the performance predictions by comparing the moving average RT over five minutes. We partitioned the measurements in intervals of ten minutes to reason on the prediction accuracy in different phases. We evaluated the response time prediction errors of all combinations of measurement runs and simulation runs, resulting in $10 \cdot 100 = 1000$ error samples. To investigate the effect of performance prediction on the degree of horizontal scaling we compared the number of scale-out operations performed per run.

1) Experiment A: Two Server Scale-Out: Experiment A restricted the maximum number of Reencoding VMs to two ($n = 2$) to isolate the transient effect of a single scale-out. The workload used in this experiment consisted of two phases. The first phase lasted 10, the second 20 minutes. In the first phase the interarrival time of users was 29 seconds. In the second phase the interarrival time decreased to 15 seconds. The lower interarrival time caused multiple re-encoding requests to overlap. This caused the average response time to increase due to increasing resource congestion in the Reencoding VM. The response time box-plots in Figure 3 illustrate that our approach significantly improved the prediction accuracy for the maximum and mean predicted RT. The prediction accuracy in Interval 2 was increased by an even larger margin. Interval 1 is not displayed as the error of both predictions is identically small until the first scale-out is executed.

Table I lists the prediction error statistics of the baseline SimuLizar, and SimuLizar extended by our approach. The consideration of scale-out execution times reduced the median prediction error from 22.5% to 16.5%. Our approach significantly reduced the median error in Interval 2 from 14.6% to 5.4%. In the other intervals, the prediction error of our extended analysis was at or below the baseline analysis.

Figure 4 shows the measured and predicted RT of the runs that were the RT medians (see Fig. 3). The baseline analysis did not consider the delayed availability of the added Reencoding instance. This is visualized by the small line in its scale-out duration segment.

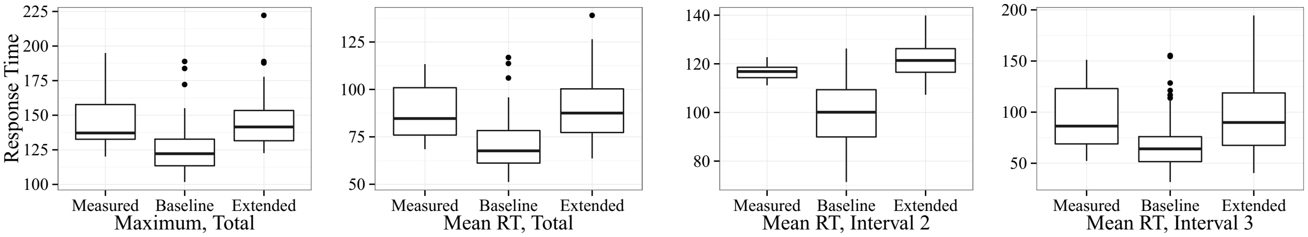

Fig. 3: RT Measurements and Simulation Results from SimuLizar Baseline and Extended by Our Approach (Experiment A)

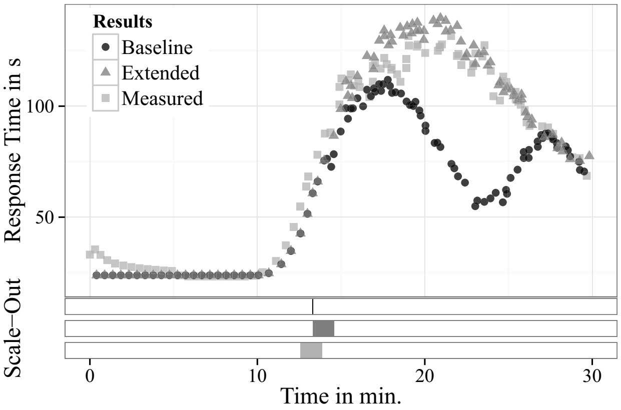

Fig. 4: Average Response Times (RTs) over 5-Minute Window and Scale-Out Actions (Experiment A). The Bars below depict the scale-out durations as rectangles covering the time when scale-out was started and finished.

TABLE II: Workload Used in Experiment B

Phase	1	2	3	4	5
Length (in min.)	10	10	10	10	10
Interarrival Time (in s)	29	15	10	15	29

2) Experiment B: Five Server Scale-Out: For Experiment B we increased the maximum scale-out factor to $n = 5$. We used the workload shown in Table II to test by which degree the application would scale with increasing load and how long it would take the system to recover.

Figure 5 displays the RT box plots for Experiment B. The consideration of the scale-out execution time improved the accuracy of the maximum and mean predicted RTs. The increase in accuracy in Interval 2 was particularly high.

Table III shows the error statistics of Experiment B. Our approach lowered the median total RT prediction error from 14.0% to 9.8%. In Interval 2, it reduced the median RT prediction error from 19.1% to 4.6%. The median prediction error in the other intervals was also at or below the prediction error of the baseline prediction.

TABLE III: Response Time Prediction Error per Workload Interval from Comparison of 10 Measurement 100 Sim. Runs

Interval	Median Error		Mean Error		Estd. Std. Dev.	
	Baseline	Extended	Baseline	Extended	Baseline	Extended
1	1.5%	1.5%	1.5%	1.5%	0.1%	0.1%
2	19.1%	4.6%	20.1%	5.6%	10.4%	4.3%
3	23.0%	23.1%	25.2%	32.5%	18.8%	31.1%
4	13.8%	13.1%	19.6%	25.4%	18.6%	47.5%
5	3.6%	2.4%	5.8%	4.7%	9.8%	5.0%
Total	14.0%	9.8%	14.6%	13.1%	7.9%	15.3%

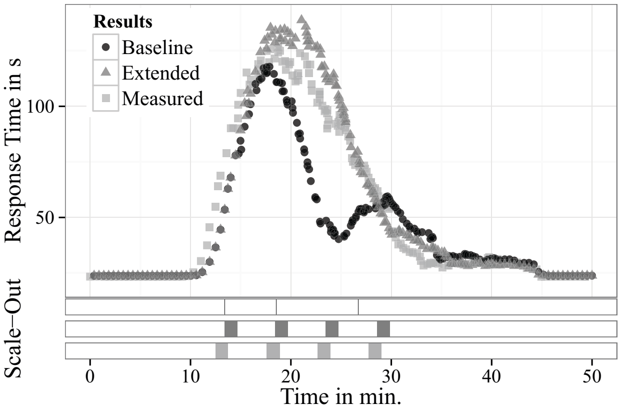

Fig. 6: Average RTs over 5-Minute Window and Scale-Out Action (Experiment B).

Figure 6 depicts the RT measurements and the scale-out execution times measured and predicted by the baseline and our approach. As is shown by the three lines in the Scale-Out section the baseline had predicted that three scale-outs would be executed. However, the deployed application had executed four scale-outs. This was correctly predicted by our approach. On average, the baseline had predicted that an average of 3.46 scale-outs would be executed. Over the ten measurements we observed the execution of 3.9 scale-outs. Our extended analysis predicted an average execution of 3.79 scale-outs. By considering the transient effects induced by scale-out, we were able to predict the number of booted VMs more accurately.

3) Conclusion: The results of Experiment A and B indicate that our approach for considering transient effects in model-driven performance analyses significantly increases the accuracy of predictions in transient phases (Q1).

D. Detecting Design Deficiencies in Self-Adaptive Systems

This section evaluates whether our approach enabled the detection of design deficiencies in self-adaptive software systems (Q2). This sections investigates whether the self-

[2] https://sdqweb.ipd.kit.edu/wiki/Transient_Effects_Case_Study

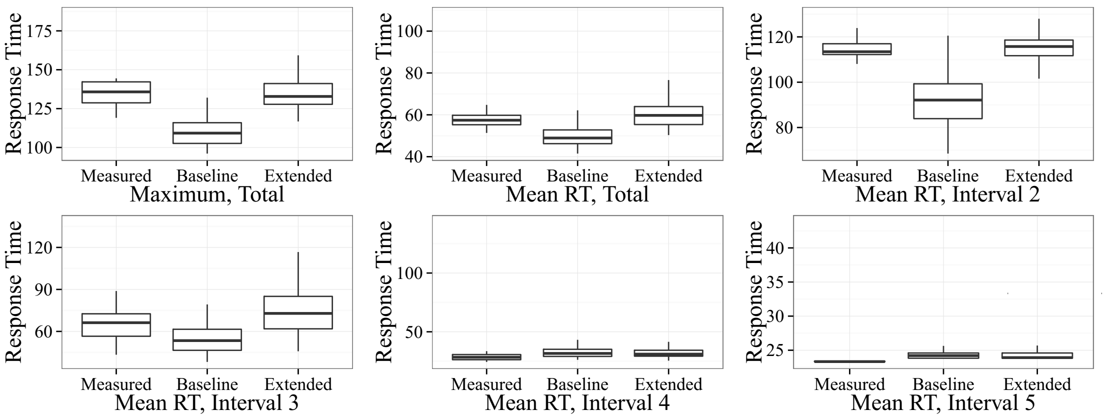

Fig. 5: RTs Measurements and Simulation Results from SimuLizar Baseline and Extended by Our Approach (Experiment B)

adaptation rule manages to uphold the requirement that the average Reencoding RT should not be higher than 120 seconds.

To predict whether the horizontally scaling Media Store would uphold our requirement under the evaluated workload, we evaluated the chosen horizontal scaling policy using SimuLizar's model-driven performance analysis. The maximum RT box plot (Fig. 5) shows that the SimuLizar baseline that the predicted average response time over five minutes did not surpass 120 seconds for the predictions in the interquartile range. Hence, the results indicated that the Horizontal Scaler would be able to maintain the response time requirement using the scalability rule presented in Section A.

The measurements of the deployed Media Store using the scale-out rule disprove the predictions. The box plot shows that the Reencoding service could not uphold the average response time requirement in over 75% of the measurement runs.

When considering the transient effects of scale-out using our approach, we were able to correctly predict that the Reencoding service would not meet the maximum RT requirement of 120 seconds. In summary, the results show that our approach enables the detection of design deficiencies of self-adaptive software system (Q2 (a)).

E. Addressing Deficiencies in Self-Adaptive Software Systems

The SimuLizar baseline wrongly predicted that the horizontal scaling rule outlined in Section A would manage to maintain an average response of less than 120 seconds. To identify a scaling rule that would maintain this requirement we applied SimuLizar extended with our approach. First, we reduced the size of the interval of condition 3 presented in Section A from 5 to 3 minutes. Second, we reduced the delay between two scale-outs to 3 minutes (condition 2). Finally, we lowered the average response time threshold of condition 3 from 52 to 30 seconds. This enabled the Horizontal Scaler component to react faster to changes in RT.

We analyzed how the changes to the conditions of the scale-out rule affected the response time using our extended analysis. Figure 7 shows box plots of the predicted maximum and average RTs for the improved scale-out rule. The predictions we performed indicated that the improved scale-out conditions would allow the Reencoding sub-system to meet the RT requirement as over 75% of simulation runs produced maximum RTs lower than 120 seconds.

Measurements on the implemented Media Store confirmed the prediction results. The box plot depicting the measurement results in Figure 3 shows that the average response time over five minutes for the updated scale-out rule did not surpass the threshold. The mean of the predicted running average RTs was 42.1 seconds which was 4.2% higher than the measured RT. The interquartile of the maximum RT was correctly predicted to be below 120 seconds.

Performing the analysis extended by our approach predicted that the requirement would not be met for the workload shown in Table II. Measurements on the deployed Media Store application confirmed the requirement violation. By considering transient effects we managed to improve the initial scale-out rule. In conclusion, the design scenario confirmed that our approach enables the detection and solution of design deficiencies in self-adaptive software systems (Q2).

F. Threats to Validity

Our evaluation investigated the accuracy and benefit gained by considering transient effects for the Media Store application deployed on an OpenStack cloud. It did not investigate the impact of transient effects in other software systems. The outlined experiments each were performed for a single workload. They did not validate whether the system would manage to maintain the performance requirements for other workloads. We conducted 10 measurement runs and compared them to 100 simulation runs per simulator. Considering a larger number of runs could theoretically affect the distribution of results.

VII. Related Work

Becker et al. have performed a survey of model-driven performance engineering approaches of self-adaptive software systems [21]. The following discusses the most relevant approaches presented in the survey as well as other work from the domains of model-driven engineering and cloud computing.

Grassi et al. [17] outline the *D-KLAPER* approach for considering dynamic reconfigurations in a model-driven approach for performance and reliability prediction. Unlike

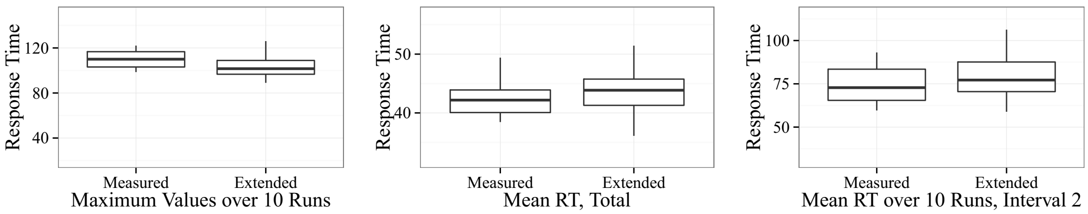

Fig. 7: RT Predictions and Measurement Results for the Refined Scale-Out Rule

SimuLizar [3], D-KLAPER only considers reconfigurations in the composition and deployment of components. D-KLAPER requires that all reconfigurations are proactively specified. Reconfigurations can not react to system events such as QoS requirement violations. Grassi et al. assume that the space of all configuration states is known prior to an analysis. D-KLAPER allows the specification of transient effects induced when transitioning between two configurations. Unlike our approach, D-KLAPER does not support the specification of transient effects independent of pre-defined system configurations.

Alansari and Bordbar [5] model VM migration policies using performance models formalized in Petri Nets. The authors estimate the cost of migration by estimating the transfer durations of VMs. These durations are, however, not included in their performance models and only estimated as part of a results analysis.

SLAstic.SIM [4] by von Massow et al. is another simulation-based performance analysis for the *SLAstic* self-adaptation framework. Unlike SimuLizar, SLAstic.SIM does not describe adaptation rules using transformations or graph patterns, but directly calls the SLAstic self-adaptation framework. SLAstic.SIM does not consider transient effects.

There exist multiple cloud simulators [6], [7] that support the evaluation of transient effects. *CloudSim* [6] simulates the behavior of cloud computing systems. Unlike model-driven performance prediction approaches like Palladio [2] and SimuLizar [3], CloudSim does not focus on evaluating the performance of the system in terms of service response times. Instead it reasons on operational efficiency based on resource utilization and consumed energy. CloudSim considers the overhead of VM migration. Unlike our approach CloudSim does not allow for architecture-level performance modeling of adaptations. Greencloud [7] is a cloud simulator focusing on network communication. Like Cloudsim, Greencloud requires that all performance effects of adaptations are implemented in code.

Many self-adaptation frameworks consider delays in the execution of self-adaptations [15], [22], [23]. As our approach allows to consider these delays among other transient effects, it enables reasoning on the effectiveness of delay-based mechanisms in dealing with transient phases. Cheng et al. [15] propose the domain-specific language *Stitch* for modeling self-adaptive software systems. Stitch follows the Strategies, Tactics and Actions (S/T/A) structuring for self-adaptive systems. It models the impact (cost and benefit) of adaptations per tactic. The impact hereby is the intended, not the actual actual effect of performing an adaptation. Stitch defines a delay per strategy and tactic. The delay is used as a deadline after which the effect of an adaptation should have been observed. As the delay is defined as a fixed duration it can only accurately account for delays caused by transient effects that are independent of the system's state and load. Consequently, it can not be used to estimate the effect of adaptations in a predictive analysis such as SimuLizar [3].

Cámara et al. [22] outline an approach for proactive self-adaptation that considers the delay with which a self-adaptation tactic is executed. Again, the authors assume that this latency is constant and can be predetermined independent of the system state. Rosa et al. [23] define the impact of individual adaptation actions on multiple quality dimensions. The authors acknowledge that the adaptation's effect on the quality dimensions depends on the context in which the action is executed. The authors address this dependency by individually defining the impact of an adaptation for each usage context and system state. Our approach does not require to specify the impact of an adaptation for each possible usage context and system state. We employ parametrized performance models that are parametrized by the usage context and system state.

Lehrig [24] eases the analysis of self-adaptive software systems introduced by Becker et al. [3] by annotating the architectural model of a software system with a higher-level description of a used adaptation mechanism. For example, a subsystem would be annotated to be scaled out based on a certain threshold. The applied threshold-based adaptation mechanism is kept in a repository for adaptation strategies. While the authors' approach eases the reuse of existing self-adaptation mechanisms for software systems, it does not address how the transient effects induced by self-adaptation mechanisms affect the system's effectiveness at achieving QoS requirements.

Performance Completions [25] enrich performance models with middleware and platform specific performance characteristics. Performance completions are defined independent of individual performance model instances. Instead, they define how the performance of a middleware component affects the performance of the context in which it is used. We employ a similar concept for the definition of our Adaptation Performance Model. Unlike the Performance Completions approach we do not refine a performance model before it is evaluated. Rather, we refine the performance model during performance analysis. This allows us to account for the performance impact of adaptation actions in system states that cannot be statically predetermined.

VIII. CONCLUSION

This paper presents an approach for considering transient effects in model-driven performance analyses of self-adaptive software systems. We formally define an Adaptation Action model that couples the description of adaptation effects, conditions and performance effects of Adaptation Actions. Our Action Model Interpreter evaluates the effects of Adaptation Actions described using our model. The interpreter can be coupled with existing model-driven performance analyses of self-adaptive software systems.

For the evaluation we coupled the SimuLizar analysis with our Action Model Interpreter. Two experiments demonstrated that our approach improved the performance prediction accuracy for the horizontally scaling Media Store application (Q1). Two evaluation experiments illustrated how the consideration of transient effects enabled us to identify design flaws in the scale-out conditions of the Media Store application (Q2).

Our approach enables software architects to detect design deficiencies of self-adaptive software systems in transient phases. By considering the performance effects of self-adaptations, the approach allows software architects to detect QoS requirement violations in critical transient phases where the user load changes.

In future work we will investigate how transient effects affect the energy efficiency of software systems. For this, we will investigate how our approach can be used to describe the energy consumption of adaptation actions such as switching between power states. Furthermore, we plan to evaluate our approach for modeling and analyzing transient effects to other types of adaptations, e.g. VM migration. To formalize reasoning on the gained benefit from considering transient effects in QoS analyses we intend to apply and evaluate techniques from the domain of time series analysis [26] to compare sets of measurement and simulation results.

Acknowledgments

This work is funded by the European Union's Seventh Framework Programme under grant agreement 610711.

References

[1] H. Koziolek, "Performance evaluation of component-based software systems: A survey," *Perform. Eval.*, vol. 67, no. 8, pp. 634–658, Aug. 2010.

[2] S. Becker, H. Koziolek, and R. Reussner, "The Palladio component model for model-driven performance prediction," *Journal of Systems and Software*, vol. 82, no. 1, pp. 3 – 22, 2009.

[3] M. Becker, M. Luckey, and S. Becker, "Performance Analysis of Self-Adaptive Systems for Requirements Validation at Design-Time," in *Proceedings of the 9th ACM SIGSOFT International Conference on Quality of Software Architectures*, ser. QoSA '13. ACM, Jun. 2013.

[4] R. Von Massow, A. Van Hoorn, and W. Hasselbring, "Performance simulation of runtime reconfigurable component-based software architectures," in *Proceedings of the 5th European Conference on Software Architecture*, ser. ECSA'11. Springer, 2011, pp. 43–58.

[5] M. Alansari and B. Bordbar, "Modelling and analysis of migration policies for autonomic management of energy consumption in cloud via petri-nets," in *International Conference on Cloud and Autonomic Computing (ICCAC)*, Sept 2014, pp. 121–130.

[6] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: A Toolkit for Modeling and Simulation of Cloud Computing Environments and Evaluation of Resource Provisioning Algorithms," *Softw. Pract. Exper.*, vol. 41, no. 1, pp. 23–50, Jan. 2011.

[7] D. Kliazovich, P. Bouvry, Y. Audzevich, and S. Khan, "Greencloud: A packet-level simulator of energy-aware cloud computing data centers," in *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, Dec 2010, pp. 1–5.

[8] M. Mao and M. Humphrey, "A performance study on the vm startup time in the cloud," in *2012 IEEE 5th International Conference on Cloud Computing (CLOUD)*, June 2012, pp. 423–430.

[9] R. H. Reussner, S. Becker, J. Happe, R. Heinrich, A. Koziolek, H. Koziolek, M. Kramer, and K. Krogmann, Eds., *Modeling and Simulating Software Architectures – The Palladio Approach.* Cambridge, MA: MIT Press, 2016, to appear.

[10] J. Kephart and D. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, Jan 2003.

[11] Object Management Group (OMG), "Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT), v1.2," February 2015. [Online]. Available: http://www.omg.org/spec/QVT/1.2/PDF

[12] M. von Detten, C. Heinzemann, M. Platenius, J. Rieke, D. Travkin, and S. Hildebrandt, "Story Diagrams - Syntax and Semantics," Software Engineering Group, Heinz Nixdorf Institute, Tech. Rep., 2012.

[13] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations," in *Model Driven Engineering Languages and Systems*, ser. LNCS. Springer, 2010, vol. 6394, pp. 121–135.

[14] A. Strunk, "Costs of virtual machine live migration: A survey," in *IEEE Eighth World Congress on Services (SERVICES)*, June 2012, pp. 323–329.

[15] S.-W. Cheng and D. Garlan, "Stitch: A language for architecture-based self-adaptation," *Journal of Systems and Software*, vol. 85, no. 12, pp. 2860 – 2875, 2012.

[16] 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*, vol. 8, no. 1, pp. 73–89, 2014.

[17] V. Grassi, R. Mirandola, and A. Sabetta, "A model-driven approach to performability analysis of dynamically reconfigurable component-based systems," in *Proceedings of the 6th International Workshop on Software and Performance*, ser. WOSP '07. New York, NY, USA: ACM, 2007, pp. 103–114.

[18] G. Franks, T. Al-Omari, M. Woodside, O. Das, and S. Derisavi, "Enhanced modeling and solution of layered queueing networks," *IEEE Transactions on Software Engineering*, vol. 35, no. 2, pp. 148–161, March 2009.

[19] A. Martens, H. Koziolek, L. Prechelt, and R. Reussner, "From monolithic to component-based performance evaluation of software architectures," *Empirical Software Engineering*, vol. 16, no. 5, 2011.

[20] C. Stier, A. Koziolek, H. Groenda, and R. Reussner, "Model-Based Energy Efficiency Analysis of Software Architectures," in *Proceedings of the 9th European Conference on Software Architecture (ECSA '15)*, ser. LNCS. Springer, 2015.

[21] M. Becker, M. Luckey, and S. Becker, "Model-driven performance engineering of self-adaptive systems: A survey," in *Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures*, ser. QoSA '12. New York, NY, USA: ACM, 2012, pp. 117–122.

[22] J. Cámara, G. A. Moreno, D. Garlan, and B. Schmerl, "Analyzing latency-aware self-adaptation using stochastic games and simulations," *ACM Transactions on Autonomous and Adaptive Systems*, 2016.

[23] L. Rosa, L. Rodrigues, A. Lopes, M. Hiltunen, and R. Schlichting, "Self-management of adaptable component-based applications," *IEEE Transactions on Software Engineering*, vol. 39, no. 3, pp. 403–421, March 2013.

[24] S. Lehrig, "Architectural Templates: Engineering Scalable SaaS Applications Based on Architectural Styles," in *Proceedings of the MODELS 2013 Doctoral Symposium*, 2013, pp. 48–55.

[25] J. Happe, S. Becker, C. Rathfelder, H. Friedrich, and R. H. Reussner, "Parametric performance completions for model-driven performance prediction," *Performance Evaluation*, vol. 67, no. 8, pp. 694 – 716, 2010, special Issue on Software and Performance.

[26] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. Keogh, "Querying and mining of time series data: Experimental comparison of representations and distance measures," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1542–1552, Aug. 2008.