SimuLizar: Design-Time Modeling and Performance Analysis of Self-Adaptive Systems*

Matthias Becker, Steffen Becker
Heinz Nixdorf Institute
University of Paderborn
{matthias.becker, steffen.becker}@upb.de

Joachim Meyer
SAP AG
joachim.meyer@sap.com

Abstract: Modern software systems adapt themselves to changing environments in order to meet quality-of-service requirements, such as response time limits. The engineering of the system's self-adaptation logic does not only require new modeling methods, but also new analysis of transient phases. Model-driven software performance engineering methods already allow design-time analysis of steady states of non-adaptive system models. In order to validate requirements for transient phases, new modeling and analysis methods are needed. In this paper, we present SimuLizar, our initial model-driven approach to model self-adaptive systems and to analyze the performance of their transient phases. Our evaluation of a proof of concept load balancer system shows the applicability of our modeling approach. In addition, a comparison of our performance analysis with a prototypical implementation of our example system provides evidence that the prediction accuracy is sufficient to identify unsatisfactory self-adaptations.

1 Introduction

Modern business information systems run in highly dynamic environments. Dynamics range from unpredictably changing numbers of concurrent users asking for service, to virtualized infrastructure environments with unknown load caused by neighboring virtual machines or varying response times of required external services. Despite such dynamics, these systems are expected to fulfill their performance requirements. In the past designers achieved this by overprovisoning hardware, which is neither cost-effective nor energy-preserving. Self-adaptation is a primary means developed over the last years to cope with these challenges. The idea is that systems react to their dynamic environment by restructuring their components and connectors, exchanging components or services, or altering their hardware infrastructure.

To deal with performance requirements of classical, non-adaptive systems, researchers developed model-driven software performance engineering approaches [CDI11]. These approaches allow early design-time performance predictions based on system models to validate performance requirements. However, classical performance engineering approaches

*Acknowledgments This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

use non-adaptive system models that do not support a dedicated self-adaptation viewpoint. Consequently, self-adaptive behavior is not considered in the prediction, and only the performance of a single system configuration, i.e., steady states, can be predicted.

The limitation on a steady state performance prediction also limits the range of analysis. For example, consider a web server system with multiple servers which is able to adapt its load balancing strategy to its actual workload. Whether this system is able to recover from an overload situation within an acceptable time or not cannot be answered when neglecting the transient phases, i.e., the self-adaptation phase in which it switches from one load balancing system configuration to another configuration. Neither can it be answered if the workload is balanced over just as many servers as really needed and is hence cost-efficient.

The contribution of this paper is SimuLizar, a modeling and model-driven performance engineering approach for self-adaptive systems. SimuLizar is based on the Palladio approach [BKR09]. It extends Palladio's modeling approach with a self-adaptation viewpoint and a simulation engine. The latter enables the performance prediction of self-adaptive systems over their various configurations, allowing the analysis of transient adaptation phases.

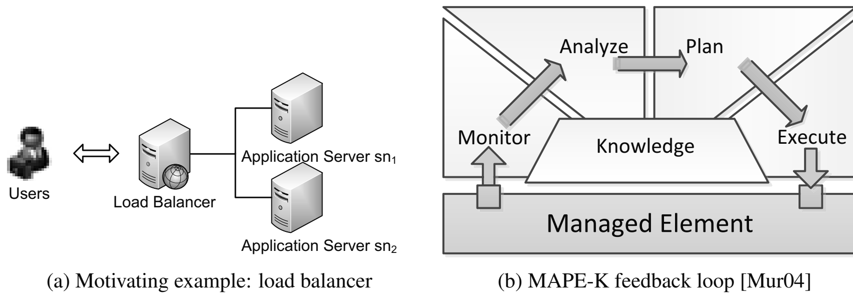
To evaluate our approach, we have applied our modeling approach to a proof of concept load balancer system. The performance analysis of the system's self-adaptation logic shows sufficiently similar characteristics as measurements taken from a performance prototype and allows us to identify unsatisfactory self-adaptation logic.

The remainder of this paper is structured as follows. We first provide a specification for a small self-adaptive load balancer in Section 2 as a motivating example. We use this load balancer system to illustrate and to evaluate the applicability of our approach. In Section 3, we briefly introduce the foundations of our work. Our SimuLizar approach is detailed in Section 4. We evaluate and discuss SimuLizar in Section 5. In Section 6, we compare our approach to related work. Finally, we conclude our work and discuss future work in Section 7.

2 Motivating Example

We provide requirements and initial design ideas for a small load balancer example system to which we will refer to throughout this paper. The load balancer we describe is not a real-life example, but serves as an understandable and easily implementable running example of a self-adaptive system.

Requirements. We want to design a load balancer system, as illustrated in Figure 1a, that distributes workload across two single-core application servers sn_1 and sn_2. The load balancer accepts client requests and delegates them to one of two application servers. Responses are sent directly from application servers to the clients. A request causes constant load of 0.3 seconds uncontended CPU time on the application server it is delegated to. We assume that we have to pay an additional load-dependent fee for utilizing the second application server sn_2. Even though the assumption of being charged load-dependently is not common for cloud services yet, our industry partners predict that this will be an option in the near future.



(a) Motivating example: load balancer (b) MAPE-K feedback loop [Mur04]

Figure 1: Motivating example and MAPE-K feedback loop.

The following requirements (R1-R3) shall be fulfilled by the load balancer system:

- **R1** The system must keep response times for user requests low. The response time for a user request must not be greater than 0.8 seconds in the mean.
- **R2** The system must keep the infrastructure costs as low as possible.
- **R3** In case R1 is not fulfilled, i.e., the mean response time is greater than 0.8 seconds, the system has to re-establish a mean response time of 0.8 seconds or less as fast as possible.

Initial Design. A software architect wants to design the load balancer such that it works properly even in high load situations. She provides initial design ideas and documents design decisions.

In order to fulfill R1, the software architect could design a load balancer to randomly delegate user request to the application servers sn_1 and sn_2. Since this conflicts with R2 the load balancer needs to preferably delegate user requests to application server sn_1 as long as R1 holds.

It is also required by R1 that the response time is less than 0.8 seconds in the mean. As defined in R3 and to save costs, only in case that requirement R1 cannot be fulfilled the load balancer delegates user request to the second application server sn_2. Regardless, it must not delegate more user requests to application server sn_2 than to application server sn_1.

In order to decide whether the mean response time is greater than 0.8 seconds, the system needs to measure the response time for each request and calculate the mean. However, it is not specified over which time span the mean should be calculated. This is again a design trade-off. Choosing a longer time span means outlier measurements are more likely ignored; choosing a shorter time span means the system detects earlier that the mean response time is greater than 0.8 seconds but also increases the monitoring overhead. The software architect chooses to calculate the mean response time from response batches within short time spans of 20 seconds. However, the software architect cannot predict whether R3 can be fulfilled yet, i.e., it cannot be predicted if the system recovers from high load situations fast enough.

In the following section, we introduce the foundations of self-adaptive systems and model-driven software performance engineering (MDSPE). Based on our example, we will create a formal system model from our initial design ideas in Section 4.

3 Foundations

First, we outline the characteristics of self-adaptive systems. Second, we introduce model-driven software performance engineering (MDSPE) and the Palladio MDSPE approach on which SimuLizar is based. We will refer to these foundations when we introduce our modeling approach and our Palladio-based performance engineering approach for self-adaptive systems in Section 4.

Modeling Self-Adaptive Systems. Self-adaptive systems are able to adapt their structure, behavior, or allocation in order to react to changing environmental situations. In our research, we focus on self-adaptive systems based on the MAPE-K feedback loop [Mur04], as illustrated in Figure 1b. The MAPE-K feedback loop consists of four steps. First, the managed element, the self-adaptive system, is *monitored* via defined sensors, e.g., for the mean response time. Second, the monitored data is *analyzed*, e.g., if the predefined threshold of 0.8 seconds is exceeded. Third, if the analysis revealed that a self-adaptation is required it is *planned*, e.g., a predefined self-adaptation rule like the load balancing rule is selected. Finally, the self-adaptation is *executed* via effectors of the managed element. In all steps, a *knowledge base* containing system information, i.e., a runtime model, can be accessed. For example, monitoring data can be stored in the knowledge base, or self-adaptation rules can be accessed from the knowledge base of the MAPE-K feedback loop.

The design and analysis of these systems is made difficult by two factors. First, the broad variety of environmental situations to which self-adaptive systems can adapt, e.g., actual workloads, requires a complex logic to monitor and analyze the environment. Second, the wide range of possible self-adaptation tactics, e.g., adapting the load balancing strategy in our motivating example, introduces additional complexity for planning and executing self-adaptive behavior.

There is an ongoing trend in software engineering to introduce a new modeling viewpoint in order to address the increased complexity of self-adaptivity [Bec11]. This new modeling viewpoint enables a separation of concerns, i.e., separating business logic from self-adaptation logic by introducing new views for modeling monitoring and reconfiguration. Thus, a dedicated analysis of requirement fulfillment of the self-adaptation logic is enabled. Our SimuLizar approach enables us to model self-adaptive systems with this new modeling viewpoint. In the next section, we provide the foundations of the performance analysis as extended by SimuLizar.

Model-Driven Software Performance Engineering. Model-driven software performance engineering is a constructive software quality assurance method to ensure performance-related quality properties [CDMI06]. Software performance properties can be quantified using several metrics, like response-time, or utilization.

Assuming that performance requirements are explicitly specified, MDSPE enables to eval-

uate whether performance requirements can be satisfied or not by a modeled software system. For this purpose a software design model is annotated with performance-relevant resource demands, such as CPU time demands. For example, in our load balancer system we know that a user request has a constant CPU demand of 0.3 seconds.

Subsequently, the annotated model is translated into analysis models or a performance prototype. Analysis models can either be simulated or solved using analytical methods, e.g., solving queuing networks with queuing theory. Performance prototypes can be deployed to the target runtime environment, i.e., servers, and performance metrics like response times can be actually measured. A correct transformation ensures that the transformed model or performance prototype is a correct projection of the software design model.

Which method of analytical solving, simulation, or performance prototyping is applied is mainly a trade-off among made assumptions and the accuracy of the performance prediction. In general, analytical solving provides accurate predictions if strict assumptions hold, e.g., the model must only include exponentially distributed processing rates and inter-arrival rates in order to be solvable. Simulation allows more relaxed assumptions but provides accurate predictions only with a high number of simulation runs. Since a performance prototype is deployable and runnable software that fakes resource consumption, the fewest assumptions have to hold for it. However, because a performance prototype needs to be deployed and executed on the target execution environments, i.e., real servers, is the most time-consuming of all performance prediction methods.

Finally, the results from analytical solving, simulation, or the performance prototype shed light on whether the performance requirements can be satisfied or not. Interpreting the results also helps revise the software design and eliminating performance flaws.

Palladio [BKR09] is an MDSPE approach for component-based software engineering that our approach is based on. One of Palladio's key features is the integrated support for design and performance analysis of component-based software. For this purpose, Palladio introduces its own component model, the Palladio Component Model (PCM), which allows us to annotate performance-relevant information in a software design model.

A PCM model consists of several artifacts, as illustrated in Figure 2. Component developers provide software components including performance-relevant properties and behavior in a *repository*. A software architect defines the *assembly* by combining components provided in the repository into a complete software system. A deployer specifies the available resource infrastructure, e.g., servers with CPUs and HDDs in a *resource environment* view. Furthermore, she specifies the concrete deployment of the system in an *allocation* view. Finally, domain experts specify typical workloads in a *usage* model.

Performance metrics, e.g., response time, of systems modeled as PCM instances can be analyzed using the Palladio tool suite. For this purpose the PCM model is automatically transformed to simulation code, performance prototypes, or analysis models.

In this paper, we focus on the two tools SimuCom [BKR09] and ProtoCom [BDH08] included in the Palladio tool suite. SimuCom is Palladio's simulation engine, which enables the simulation and performance analysis of systems modeled with PCM. Our SimuLizar approach reuses the functionality provided by the SimuCom engine and extends it to simulate self-adaptive systems. ProtoCom is a tool to transform PCM instances into perfor-

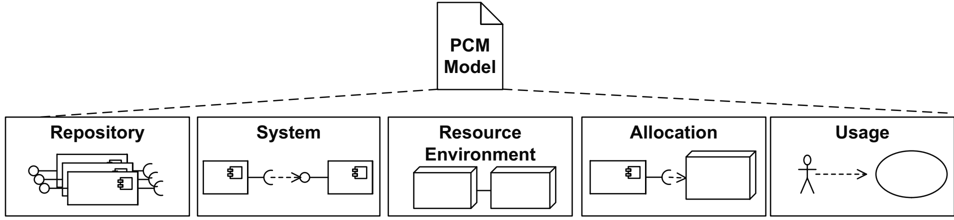
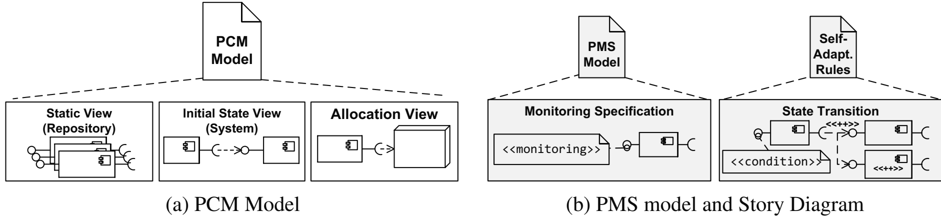


Figure 2: Palladio Model with all artifacts.



(a) PCM Model (b) PMS model and Story Diagram

Figure 3: Modeling views: (a) PCM mapped to proposed views and (b) newly implemented views.

mance prototypes. We use ProtoCom to generate a performance prototype. Subsequently, we extend the generated prototype with self-adaptive functionality, in order to validate the correctness of the performance predictions of our SimuLizar approach.

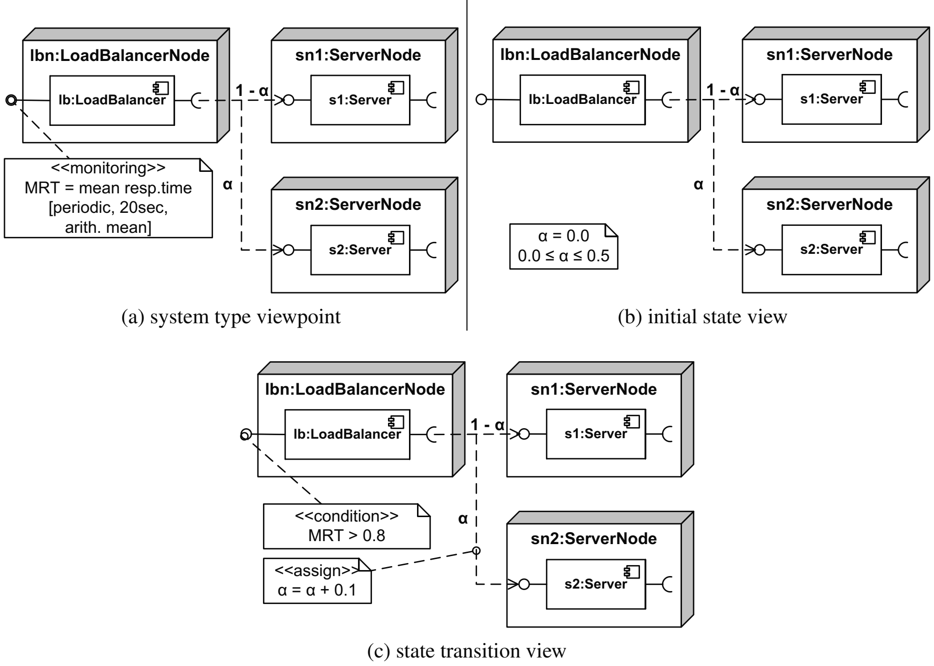
4 SimuLizar

SimuLizar extends Palladio in two areas: (a) the modeling and (b) design-time performance analysis of self-adaptive systems.

Modeling Approach. SimuLizar provides a modeling approach for self-adaptive systems based on ideas presented in [Bec11]. In SimuLizar, a self-adaptive system model consists of two viewpoints with several views each. First, a system type viewpoint consisting of three views: a *static view*, a *monitoring specification view*, and an *allocation view*. Second, a runtime viewpoint including the *initial state view* and the *state transition view*.

Where appropriate, we have mapped the required views to existing artifacts of PCM, as illustrated in Figure 3a. The static view is covered by the PCM repository, the initial state view can be mapped to the PCM system view, and the allocation is covered by PCM's allocation view.

Until now, PCM did not offer modeling views for the monitoring view or the state transition view. To fill these gaps, we introduce two new artifacts in SimuLizar, as illustrated in Figure 3b. First, the Palladio Measurement Specification (PMS), a domain-specific language for the monitoring specification view. Second, self-adaptation rules for specifying the state transition view.



(a) system type viewpoint

(b) initial state view

(c) state transition view

Figure 4: (a) System Type View, (b) Initial State View, and (c) Transition state view.

With PMS, sensors for a self-adaptive system can be specified. A monitor specification consists of four characteristics: (1) a sensor location, (2) a performance metric type, (3) a time interval type, and (4) a statistical characterization. The sensor location specifies the place of the sensor within the system, for example a service call. Currently, we support the following performance metric types: waiting time, response time, utilization, arrival rate, and throughput. Each of these metrics can be measured at the specified sensor location in one of the three time interval types: in periodic time frames with the length Δt beginning at time 0.0, periodic time frames with length Δt and a delay of Δd from the start time, or within a single fixed time frame with start time t_{start} and end time t_{stop}. Finally, for each sensor a statistical characterization can be specified to aggregate the monitored data. We support the modes *none*, *median*, *arithmetic mean*, *geometric mean*, and *harmonic mean*.

Self-adaptation rules consist of a condition and a self-adaptation action. A condition has to reference a sensor and to provide a boolean term. If the boolean term evaluates to true, the self-adaptation action is triggered. The self-adaptation action part references elements in the PCM model. Variables of these elements can be set using the ≪*assign*≫ keyword, or new instances of model elements can be instantiated using the ≪++≫ keyword. We use Story Diagrams [vDHP+12] to formalize the condition as well as the self-adaptation action as described above.

Figure 4 illustrates a model of our motivating example using our SimuLizar modeling approach. We model the system according to our initial design ideas in Section 2. The system type viewpoint, Figure 4a, uses the PCM allocation view annotated with measurement

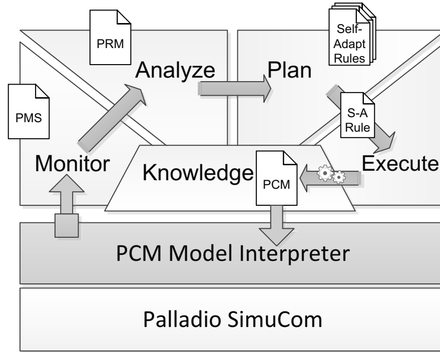


Figure 5: SimuLizar architecture.

specifications, which are defined via the PMS. In this example, the component LoadBalancer is deployed on a node lbn. The LoadBalancer component is connected to two Server components via its required interface. The two server nodes are deployed on two different ServerNodes sn_1 and sn_2. The monitoring annotation connected to the provided interface of the LoadBalancer component specifies a monitor "MRT" for the mean response time (arithmetic mean) in periodic time frames with the length of $\Delta t = 20\ seconds$. On one hand, this should reduce the monitoring overhead and prevent the system to reconfigure every time a response exceeds the target response-time of 0.8 seconds. On the other hand, it is necessary to detect whether R1 is fulfilled. The initial state view, Figure 4b, illustrates the initial configuration of the system. The variable α denoting a branch probability is set to 0.0, meaning that the whole workload is initially handled by server node sn_1 (R2). In the state transition view, Figure 4c, the variable α is set to $\alpha + 0.1$ if the condition MRT > 0.8 holds. This reflects the system's reconfiguration to satisfy R3.

Simulation Tool. Self-adaptive systems modeled with SimuLizar can be simulated using SimuLizar's MDSPE tool for self-adaptive systems, e.g., in order to get response time predictions of the overall system. The building blocks of this tool can be mapped to the MAPE-K feedback loop, as illustrated in Figure 5. In SimuLizar the managed element is the modeled simulated self-adaptive system. The simulated system is monitored, the monitoring results are analyzed, reconfiguration is planned, and a reconfiguration is executed on the simulated system if required. Reconfigurations are model-transformations of this PCM model.

A *PCM model interpreter* traverses the PCM model for each single user request, as specified in the PCM usage view. It utilizes the simulation engine, *SimuCom*, for simulating the user and system behavior including the simulation of resources. During the simulation, the simulated system is *monitored* and measurements are taken as specified via *PMS*. Whenever the PCM model interpreter arrives at a monitor place, it simulates a measurement as specified in the PMS model. Once new measurements are available the runtime model of the system, the Palladio Runtime Measurement Model (*PRM*), is updated with the newly taken measurements. An update of the PRM, i.e., new measurements, triggers the *planning* phase.

In the planning phase all *self-adaptation rules* are checked. If the condition of a rule holds, the corresponding self-adaptation action is *executed*, i.e., the PCM model is transformed.

During the execution phase, the translated model-transformation is applied to the PCM model. Subsequently, the interpreter uses the transformed PCM model / reconfigured system for new users.

5 Evaluation

To evaluate the applicability of SimuLizar we model the self-adaptive load balancer, as presented in Section 4, and analyze its performance. First, we explain how we conducted the evaluation. Second, we present the results of SimuLizar's predictions. We discuss the quality of SimuLizar's predictions compared to the measurements taken at the performance prototype. Subsequently, we discuss the possibilities to reason about transient states of self-adaptive systems. Finally, we point out the limitations of our SimuLizar approach.

Conducting the Evaluation. We evaluate SimuLizar according to four criteria. First, we validate whether our modeling approach for self-adaptive systems is sufficient to model the important aspects of self-adaptive systems (**C1**). Our second criterion is whether SimuLizar allows us to predict the performance of a self-adaptive system in the transient phases (**C2**). Third, we evaluate whether SimuLizar's predictions do not significantly deviate from the performance of a self-adaptive system performance prototype, i.e., it can be used to evaluate design alternatives (**C3**). The fourth criterion for our evaluation is whether SimuLizar's performance prediction helps to identify unsatisfactory self-adaptation logic which traditional MDSPE did not aim at (**C4**).

In order to evaluate C1, we model the load balancer system[1] using our SimuLizar modeling approach as described in the previous section. Next, we simulate the load balancer and predict its performance using SimuLizar's MDSPE tool to evaluate C2. Specifically, we simulate the system to initially start in a high load situation in order to trigger the system's self-adaptation. We expect the system to self-adapt immediately after it detects the high load situation until the response times decrease.

In order to evaluate C3, we compare SimuLizar's performance prediction to the performance predictions from a generated and manually extended performance prototype. For this, we generate the performance prototype from the load balancer model using Palladio's ProtoCom tool. We extend this prototype with a MAPE-K feedback loop to add self-adaptive behavior.

We analyze the response times for both the system model simulated with SimuLizar's MDSPE tool as well as the ProtoCom performance prototype in the high load situation. Both, SimuLizar's MDSPE tool and ProtoCom, are configured to take 1000 measurements, i.e., simulate/measure 1000 user requests. We calibrated our prototype system's resource demands according to the model, i.e., a user request has a CPU demand of 0.3 seconds in

[1]The full model can be obtained via http://goo.gl/O480s (anonymous user access)

the prototype system as well. We expect that the self-adaptive system is able to recover from the high load situation within the time of 1000 measurements.

To evaluate C4, we compare the performance predictions of SimuLizar to a series of steady state analyzes. For this, we designed a queuing network and simulate it with JSIMgraph[2]. Our expectation is, that our modeled load balancer system eventually self-adapts itself such that its configuration is in a state in which a steady state analysis implies that R1, response time lower than 0.8 seconds, is fulfilled. We configure JSIMgraph to calculate response times and utilizations for ServerNodes sn_1 and sn_2; we set the confidence interval to 99% with a maximal relative error of 3%.

Results. We first present SimuLizar's performance prediction results and the measurements taken with the performance prototype. Second, we present JSimgraph's steady state analysis results for all possible configurations of the evaluation system.

Figure 6 shows the (interpolated) time series of response times of our specified evaluation usage scenario. The time series show that initially all requests are answered by application server sn_1. Both SimuLizar and the performance prototype measurements, show steadily increasing response times. Hence, application server sn_1 is in a high load situation. Approximately after 20 seconds the first requests are answered by application server sn_2 in both series. This indicates that the load balancer has triggered a self-adaptation, which is further confirmed by plateaus in the increase of the response times from application server sn_1. In both time series, we can finally observe that the system triggers self-adaptation five times. Only after the several self-adaptation, approximately after 80 seconds, the responses times of both server ServerNode sn_1 and ServerNode sn_2 have similar values. Any deviations of the predicted and measured response times occur due to the random load generation and balancing strategy of our evaluation system.

We conclude from these results that self-adaptive systems can be modeled (C1) and their performance within transient phases can be predicted (C2). Furthermore, both time series do not deviate significantly, i.e., less than 30%, from each other. In the performance engineering community such a deviation is considered sufficient to differentiate between design alternatives (C3).

To evaluate C4, Figure 7a shows the queuing network we have simulated with JSIMgraph. The source spawns new users with the same rate as the load balancer example model. It routes users with the probability of $1 - \alpha$ to server sn_1 and with probability α to server sn_2. Server sn_1 and Server sn_2 are both queues with unlimited queue size and a constant service time of 0.3s.

The results for mean response time and mean utilizations of the JSIMgraph simulation are denoted in Figure 7b. Each row represents one configuration of the system, i.e., different values for α. When $\alpha = 0.0$ the mean response time is 4.421 seconds. The utilization of server sn_1 is 0.9299, i.e., 93%, which indicates a high load situation. The utilization of server sn_2 is 0%, because no users are routed to it.

Interestingly, the steady state analysis shows, that the mean response time with $\alpha = 0.3$ is 0.759 seconds, which falls below our threshold of a mean response time of 0.8 seconds.

[2]JSIMGraph is included in the Java Modeling Tools which can be obtained via http://jmt.sourceforge.net/

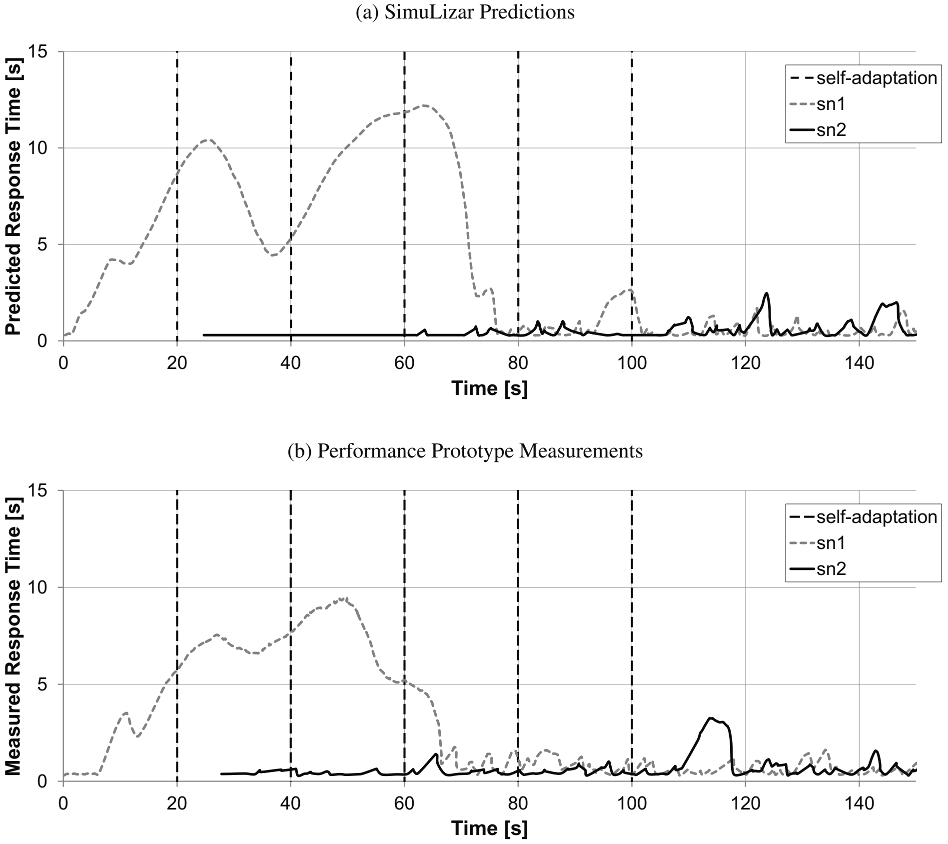
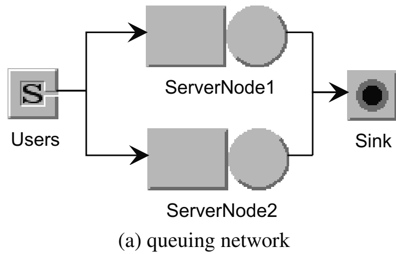


Figure 6: The (interpolated) time series for the response times of (a) SimuLizar's simulated system and (b) measured response times for the performance prototype. The vertical axes represent response times of a single user request. The horizontal axes represent the execution/simulation time. Each curve represents the response times of a series of single user request. The dashed grey curve represent response times for user request delegated to application server sn_1; the solid black curve represent response times for user requests delegated to application server sn_2.



(a) queuing network

α	Mean RT	Mean Utilization	
	System	Server 1	Server 2
0.0	4.421	0.9299	0.0000
0.1	1.743	0.8363	0.0932
0.2	1.042	0.7513	0.1858
0.3	0.759	0.6537	0.2786
0.4	0.628	0.5563	0.3600
0.5	0.580	0.4677	0.4664

(b) response times and utilizations

Figure 7: Evaluation system represented as (a) queuing network and (b) steady state analysis for each state.

Nevertheless, we can observe that our evaluation system triggers two more self-adaptations until $\alpha = 0.5$. From this, we can conclude that the effects of setting α to 0.3 do not immediately effect the mean response time to fall below the threshold. Even in the next mean response time batch the threshold is exceeded and another self-adaptation is triggered.

As these queuing network shows, in this scenario adapting α to at least 0.3 right from the start would help to recover the system from the overload situation faster. This means that our adaptation rules violate R3 because the system does not recover as fast as possible. Hence, we identified an unsatisfactory self-adaptation logic (C4).

Limitations. SimuLizar as well as our evaluation still underlie several assumptions and limitations. First, SimuLizar cannot be considered as sufficient for a comprehensive performance prediction of self-adaptive systems yet. Second, our evaluation is limited due to some threats to validity.

Self-adaptive behavior is usually triggered to cope with environmental changes, e.g., changing customer arrival rates. However, the PCM usage view we are using here is static, i.e., a dynamically or randomly changing environmental cannot be modeled. Thus, we are forced to model usage scenarios in which the condition of a self-adaptation rule holds from the start. This limits the scope of a simulation to only a set of rules which are triggered for the modeled static usage scenario.

Due to the fact that SimuLizar simulates self-adaptive systems, it underlies some assumptions with respect to the self-adaptation of a system. In SimuLizar and self-adaptive system models we neglect resource consumption of self-adaptations. Furthermore, we do not handle exceptions that might occur during the self-adaptation.

Our evaluation is mainly limited due to the implementation of our performance prototype. First, the measurements taken from the performance prototype rely on a prior calibration of the machine it runs on. Although this calibration has been tested and evaluated before, the actual generated load is subject to minor deviations, e.g., due to system background load. This leads to inaccurate measurement results and may bias our evaluation. Second, since our performance prototype is a distributed system, asynchronous clocks are another threat to the validity of the taken measurements and our evaluation. Third, the usage we specified for our evaluation example contains an exponentially distributed arrival rate of

the customers. Hence, the arrivals are random and are consequently not generalizable, i.e., the actual inter-arrival rate may vary in each evaluation run for both, the simulation and the performance prototype.

6 Related Work

In recent years, several related articles about self-adaptive system modeling and model-driven performance engineering have been published. We have surveyed them in [BLB12].

In [FS09], Fleurey and Solberg propose a domain-specific modeling language and simulation tool for self-adaptive systems. With the provided modeling language self-adaptive systems can be specified in terms of variants using an EMF-based meta-model. Furthermore, functional properties of the specified system can be checked and simulations can be performed. SimuLizar's focus is on the performance aspect of self-adaptation in contrast to the approach by Fleurey and Solberg which focuses on functional aspects.

The D-KLAPER approach [GMR09] is an MDSPE approach which can be applied to self-adaptive systems. D-KLAPER does not aim at providing modeling support for software design models. Instead software design models annotated with performance-relevant resource demands have to be transformed into D-KLAPER's intermediate language. D-KLAPER then provides the necessary tools to analyze a self-adaptive system specification provided in the intermediate language. However, in contrast to our focus on transient phase analysis, systems analyzed with D-KLAPER are considered to be in a steady state when analyzed.

Huber et al. [HvHK+12] present a modeling-language for specifying self-adaptive behavior in the context of software design models. Their work is integrated within the Descartes project, which envisions self-adaptive cloud systems enhanced with runtime performance analysis. The focus of the Descartes project is on runtime adaptation in contrast to our aims of design-time performance analysis of self-adaptation.

QoSMOS [CGK+11] is an approach to model and implement self-adaptive systems whose self-adaptation is driven by performance requirements. The system designer has to manually derive an analysis model from the architectural model. Manually derived analysis model serve as an initial input for the online performance analysis and its parameters are updated at run-time. In contrast, we provide a full tool chain for design-time performance analysis of self-adaptive systems that also able to adapt their structure.

7 Conclusion

In this paper, we presented SimuLizar, a model-driven software performance prediction approach for self-adaptive systems. We implemented our previously introduced modeling approach and provide a simulation tool for performance prediction.

Our evaluation shows that SimuLizar is applicable to model self-adaptive systems and pre-

dict their performance. Furthermore, SimuLizar enables the analysis of transient phases during a system's self-adaptation, and allows to identify unsatisfactory self-adaptation logic.

We are working towards further enhancements of our approach. First, we plan to provide a domain-specific modeling language for specifying dynamic system workloads, i.e., randomly changing loads of simulated users. Second, we are currently implementing automatic generation of a performance prototype with self-adaptation capabilities from models specified with SimuLizar. Finally, we plan to enhance our tool to support developers modeling and evaluating self-adaptive systems, i.e., automatically detecting self-adaptation design flaws and providing design alternatives.

References

[BDH08] S. Becker, T. Dencker, and J. Happe. Model-Driven Generation of Performance Prototypes. In S. Kounev, I. Gorton, and K. Sachs, editors, *Performance Evaluation: Metrics, Models and Benchmarks*, volume 5119 of *LNCS*, pages 79–98. Springer, 2008.

[Bec11] S. Becker. Towards System Viewpoints to Specify Adaptation Models at Runtime. In *Proceedings of the Software Engineering Conference, Young Researches Track (SE 2011)*, volume 31 of *Softwaretechnik-Trends*, 2011.

[BKR09] S. Becker, H. Koziolek, and R. Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3–22, 2009.

[BLB12] M. Becker, M. Luckey, and S. Becker. Model-Driven Performance Engineering of Self-Adaptive Systems: A Survey. In *Proc. of the 9th ACM SigSoft Int. Conf. on Quality of Software Architectures*, June 2012.

[CDI11] V. Cortellessa, A. Di Marco, and P. Inverardi. *Model-Based Software Performance Analysis*. Springer, 2011.

[CDMI06] V. Cortellessa, A. Di Marco, and P. Inverardi. Software performance model-driven architecture. In *Proc. of the 2006 ACM symposium on Applied computing*, SAC '06, pages 1218–1223, 2006.

[CGK+11] R. Calinescu, L. Grunske, M. Kwiatkowska, R. Mirandola, and G. Tamburrelli. Dynamic QoS Management and Optimization in Service-Based Systems. *IEEE Trans. on Software Engineering*, 37:387–409, 2011.

[FS09] F. Fleurey and A. Solberg. A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. In *Model Driven Engineering Languages and Systems*, volume 5795 of *LNCS*, pages 606–621. Springer, 2009.

[GMR09] V. Grassi, R. Mirandola, and E. Randazzo. Model-Driven Assessment of QoS-Aware Self-Adaptation. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pages 201–222. Springer, 2009.

[HvHK+12] N. Huber, A. vn Hoorn, A. Koziolek, F. Brosig, and S. Kounev. S/T/A: Meta-Modeling Run-Time Adaptation in Component-Based System Architectures. In *9th IEEE Int. Conf. on e-Business Engineering (ICEBE 2012)*, September 9-11 2012.

[Mur04] R. Murch. *Autonomic Computing*. IBM Press, 2004.

[vDHP+12] M. von Detten, C. Heinzemann, M. C. Platenius, J. Rieke, D. Travkin, and S. Hildebrandt. Story Diagrams Syntax and Semantics. Technical Report tr-ri-12-324, Software Engineering Group, Heinz Nixdorf Institute, July 2012. Ver. 0.2.