Model-based Autonomic and Performance-aware System Adaptation in Heterogeneous Resource Environments: A Case Study

Nikolaus Huber *, Jürgen Walter *, Manuel Bähr †, Samuel Kounev *
* University of Würzburg, Chair of Software Engineering, 97074 Würzburg, Germany
nikolaus.huber@uni-wuerzburg.de, juergen.walter@uni-wuerzburg.de, samuel.kounev@uni-wuerzburg.de
† Blue Yonder GmbH & Co. KG., 76139 Karlsruhe, Germany, manuel.baehr@blue-yonder.com

***Abstract*—Recent trends like cloud computing show that service providers increasingly adopt to modern self-adaptive system architectures promising higher resource efficiency and lower operating costs. In this paper, we apply a holistic model-based approach to engineering performance-aware system adaptation. More specifically, we employ the Descartes Modeling Language (DML), a domain-specific language for modeling the performance behavior and run-time adaptation processes of modern dynamic IT systems. The conducted case study evaluates the applicability and effectiveness of our approach and demonstrates that DML provides suitable modeling abstractions that can be used as a basis for self-adaptive performance and resource management in heterogeneous environments. We apply a holistic model-based approach to build a self-adaptive system that automatically maintains performance requirements and resource efficiency in the heterogeneous resource environment of Blue Yonder. The application of DML enables to automatically adapt service infrastructures to changing customer workloads and service-level agreements in heterogeneous environments.**

I. Introduction

Recent trends like cloud computing and virtualization show that more and more service providers are adopting self-adaptive system architectures providing increased flexibility and dynamics. They are driven by the pressure to improve the efficiency of their systems, e.g., by sharing resources, and to reduce their operating costs. However, to achieve increased resource efficiency, the system must be adapted continuously to changes in the system environment. For example, the amount of resources allocated to each service must be continuously adjusted to match the changing resource demands resulting from variations in the customer workloads. The challenge is how to perform such system adaptations in an autonomic manner and at run-time without disturbing the system operation.

To address this challenge, many researchers in the autonomic computing and self-adaptive systems communities are working on the systematic engineering of self-adaptive software systems [7]. A promising approach to manage the complexity of engineering such systems is to apply model-driven techniques as the models can provide valuable information to support adaptation decisions and the automation of decision making [3]. However, current approaches in the area of model-based performance engineering of self-adaptive systems neglect modeling the dynamic context of systems, explicitly [2]. Addressing this aspect, our prior work includes the Descartes Modeling Language (DML) [**?**] and the S/T(A (Strategies/Tactics/Actions) framework [12]. DML is a novel architecture-level modeling language for modeling quality-of-service (QoS) and resource management related aspects of modern dynamic IT systems, infrastructures and services. In connection with the adaption framework S/T/A a full end-to-end adaption process can be modeled and automated. However, the approach has never been applied in a realistic scenario. In [12], we initially sketched our approach on an artificial benchmark running on homogeneous hardware environment.

In this paper, we prove the real world applicability of our approach holistic model-based approach by extension and application in a realistic case study. In concrete, we address real-world performance and resource management problems of Blue Yonder, a leading service provider in the field of predictive analytics and big data. For Blue Yonder it is business critical to estimate how much resources are required to sustain the workloads of their customers. Blue Yonder‘s current manual resource provisioning uses dedicated resources for each customer to fulfill their respective SLAs. The results of this paper enable Blue Yonder to automate the resource provisioning process and to improve resource efficiency via shared resources while ensuring the SLAs.

For this case study, we first create a model of the Blue Yonder system using DML as modeling formalism. We then define an adaptation process at the model level that adapts the system to changing customer workloads and performance requirements while considering Blue Yonder’s heterogeneous resource environment composed of low-cost desktop computers and high-end machines. The goal of the case study is to demonstrate that DML provides suitable modeling abstractions to describe the performance behavior and the degrees of freedom of Blue Yonder’s system, as well as to specify an adaptation process that can effectively maintain different performance requirements and resource efficiency at run-time. The contributions of this paper are: i) a refinement of the classical MAPE-K adaption approach entirely based on architectural performance models ii) an exemplary end-to-end application of our holistic adaptation approach to a real-life system with representative settings iii) an experimental evaluation of our model-based adaptation process demonstrating its

effectiveness and practical applicability.

The paper is structured as follows: In Section III, we introduce the major concepts of the DML and show how we integrate it into a framework to realize autonomic performance-aware resource management. In Section IV, we first present the performance and adaptation process models we created for the Blue Yonder system. Then, we present and discuss the evaluation results of our case study. In Section II we discuss related work and conclude the paper in Section V.

II. Related Work

According to [6], essential elements for building self-adaptive systems are feedback loops, as they provide a generic mechanism for self-adaptation. The software engineering community conceptually distinguishes COLLECT, ANALYZE, DECIDE, and ACT [7] whereas the autonomic computing comunity refers to MAPE-K (Monitor, Analyze, Plan and Execute based on Knowledge) [?] to describe the adaption on an abstract level. With the adoption of virtualization and cloud computing, many approaches for managing performance properties and resource efficiency have been published, e.g., [14], [18], [20], [25], [28]. These approaches normally use predictive performance models like queuing networks, layered queuing networks, queuing Petri nets, stochastic process algebras, and statistical regression models. However, the shortcoming of predictive performance models is that these models do not explicitly model architectural information that can be leveraged for specifying adaptation processes and improving automated adaptation decisions. Although there already exist approaches to model the performance behavior of software systems at the architecture level [17], such approaches are unsuitable for the model-driven performance engineering of self-adaptive systems as they neglect the dynamic context of systems or do not provide decision support for adaptation strategies [2].

If we extend our scope and consider other QoS attributes besides performance, we can find different approaches on self-adaptive software as surveyed in [22]. In this area, examples for approaches that explicitly use architectural models are [21], [10]. In [21], the authors examine the role of software architecture in self-adaptive systems and present an approach for self-adaptive software based on architectural models. In this approach, a software system is described as a dynamic architecture, characterized as a graph of components and connectors and architectural changes are regarded as graph-rewrite operations. Another approach is Rainbow [10], a framework that supports the development of self-adaptation capabilities, based on an abstract architectural model. As in the previous approach, the framework and the architectural model are generic and do not explicitly model performance-related properties that can be leveraged for autonomic performance and resource management at run-time.

Regarding the specification of adaptation actions and processes at the model level, there exist approaches that use well-known methods from the area of graph grammars. An example is Story Diagrams, a graph grammar language based on UML and Java that can be used to model the evolution and dynamic behavior of complex object structures [9]. Another example is [1], also a UML-based approach to adapt architectural models using graph transformations. Although such approaches support the specification of adaptations to models, they are not designed for the specification of adaptation processes at the model level to support engineering of self-adaptive systems. To bridge this gap, the authors of Rainbow developed Stitch [8], a programming language-like notation that can be used to describe architecture-based repair strategies. Although this approach provides useful concepts for describing adaptation processes, it does not support the specification of adaptation processes based on architecture-level performance models. A generic, model-driven approach for engineering adaptation engines is [26]. This approach provides a domain-specific modeling language, EUREMA, to support the development of adaptation engines by modeling the feedback loops of the software at a higher level of abstraction. However, this approach is targeted at engineering self-adaptive software in general, whereas the approach we use in this case study is focused on performance and resource management specific aspects of the adaptation.

III. Model-Based Performance and Resource Management

The model-based approach we apply in this work consists of two major building blocks. The first is the Descartes Modeling Language (DML), a novel modeling language to describe the quality-of-service (QoS) properties of modern IT systems as well as their dynamic context and adaptation process. The second building block is a control loop in which we employ the DML to achieve model-based self-adaptation. In Section III-A, we present a brief overview of DML. More details about the meta-modeling concepts can be found in [5], [11], [12]. In Section III-B, we explain the model-based adaptation control loop.

A. Overview of the Descartes Modeling Language

The Descartes Modeling Language is provided as a set of meta-models with a modular structure, separated into several sub-meta-models according to the major aspects relevant for modeling performance and resource management of modern IT systems. For these sub-models, which match to the headings of the following subsections, Figure 1 provides an overview and Figure 4 depicts an exemplary application.

The first aspect is the execution environment of the system comprising its infrastructure and platform resources and their distribution. This can be modeled with the *resource landscape* meta-model introduced in Section III-A1. Two other important aspects are the software system's design and implementation as well as the external services it uses. Such aspects can be described with the *application architecture* meta-model presented in Section III-A2. Finally, the deployment of the hosted applications on the infrastructure (Section III-A3) and the way the system is used (usage profile, Section III-A4) also influence the performance and resource efficiency of

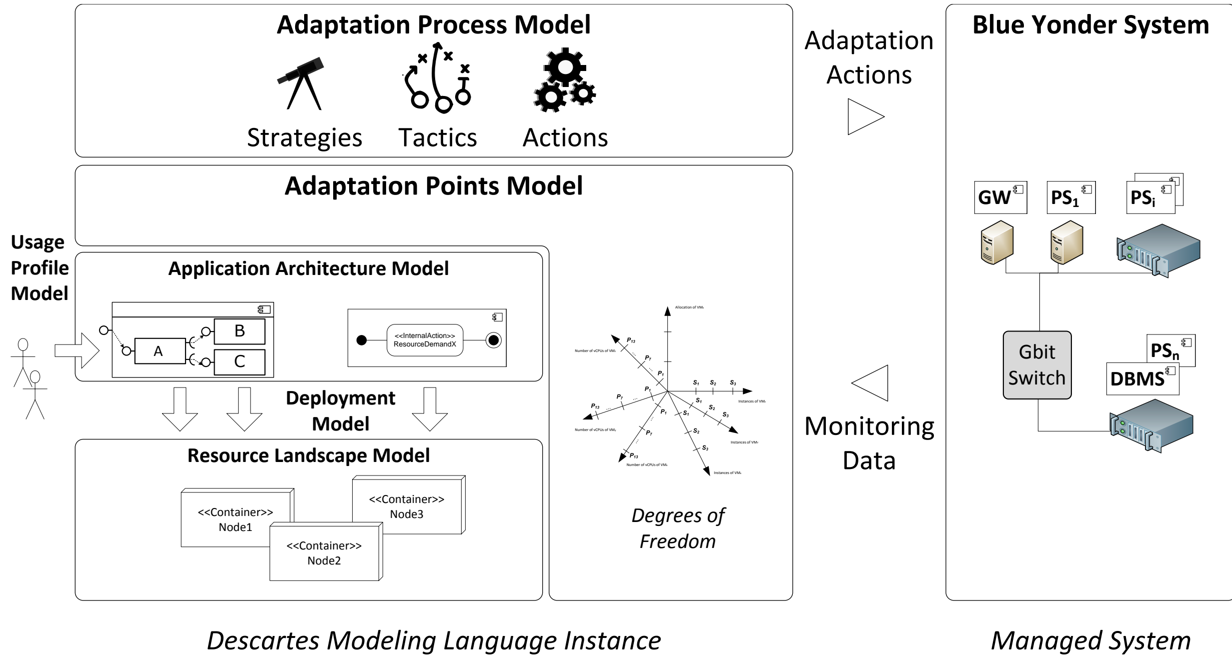

Fig. 1: Structure of a Descartes Modeling Language (DML) instance and its relation to the Blue Yonder system.

the system. Together, these models form an architecture-level performance model, i.e., they capture the properties of the system that are relevant for performance analysis at the architecture-level [16]. Such analyses can be used for reasoning at run-time to guide system adaptation processes. Thus, this model can also be compared to a *reflective* model [27]. Furthermore and in advance to existing architectural performance modeling approaches, DML provides models to capture the system's adaption points and adaption processes which are both essential for self-adaptive performance and resource management at run-time. The system's degrees of freedom, i.e., the points where the modeled system can be adapted as well as its possible valid system states are captured by the *adaptation points* meta-model, described in Section III-A5. The adaptation processes themselves, i.e., the way the system adapts to changes in the environment, are described with the *adaptation process* meta-model, presented in Section III-A6.

1) Resource Landscape: The purpose of the *resource landscape* meta-model is to describe the structure and properties of both physical and logical resources of modern distributed IT service infrastructures. Therefore, the resource landscape meta-model provides modeling abstractions to specify the available physical resources (CPU, network, HDD, memory) as well as their distribution within data centers (servers, racks, etc). Furthermore, the resource landscape meta-model also supports modeling the layers of the execution environment and specifying the performance influences of the configuration of each layer. In this context, resource layers represent layers of the software stack on which software is executed like virtualization, operating system, middleware, and runtime environments (e.g., JVM). In addition, the meta-model also considers the distribution of resources across data centers. More details about the resource landscape meta-model can be found in [11].

2) Application Architecture: We model the application architecture of the managed system according to the principles of component-based software systems. A software *component* is defined as a unit of composition with explicitly defined provided and required interfaces [24]. To describe the performance behavior of a service offered by a component, the *application architecture* meta-model supports multiple (possibly co-existing) behavior abstractions at different levels of granularity. The behavior descriptions range from a *black-box* abstraction (a probabilistic representation of the service response time behavior), over a *coarse-grained* representation (capturing the service behavior as observed from the outside at the component boundaries, e.g., frequencies of external service calls and amount of consumed resources), to a *fine-grained* representation (capturing the service's internal control flow, considering performance-relevant actions). The advantage of the support for multiple abstraction levels is that the model is usable in different online performance prediction scenarios with different goals and constraints, ranging from quick performance bounds analysis to detailed system simulation. Moreover, one can select an appropriate abstraction level to match the granularity of information that can be obtained through monitoring tools at run-time, e.g., considering to what extent component-internal information can be obtained by the available tools. Further details including a complete specification of the application architecture meta-model can be found in [5].

3) Deployment: To capture the interactions between the resource landscape and the application architecture, one must

model the connection between hardware and software using the *deployment* meta-model. It associates software component instances of the application architecture meta-model with container instances of the resource landscape meta-model.

4) Usage Profile: To model user interactions with the system (i.e., the usage profile), DML provides a *usage profile* meta-model. A usage profile model contains one or more usage scenarios that can be seen as a combination of UML use cases and UML activities. A usage scenario describes the workload type (e.g., open or closed workload), the workload intensity (e.g., request arrival rates or think times), and the user behavior, i.e., which services are called and in what sequence. Further details can be found in [11].

5) Adaptation Points: The *adaptation points* meta-model is an addition to the resource landscape and application architecture meta-models providing modeling constructs to describe the elements of the resource landscape and the application architecture that can be adapted (i.e., reconfigured) at run-time (see Figure 4 for examples). Other model elements that may change at run-time but cannot be directly controlled (e.g., the usage profile), are not in the focus of this meta-model. The same holds for the adaptation process itself, i.e., how the system is adapted in a given online scenario is specified in the adaptation process meta-model described in Section III-A6. In the terminology of [27], this model is an instance of the change model. A more detailed description of the adaptation points meta-model can be found in [11].

6) Adaptation Process: The *adaptation process* meta-model describes the process that keeps the system in a state such that its operational goals are continuously fulfilled, i.e., it describes the way the system adapts to changes in its environment. The meta-model consists of three main elements used to describe the adaptation process at three different levels of abstraction. At the top level are the *strategies* where each strategy aims to achieve a given high-level objective. A strategy uses one or more *tactics* to achieve its objective. Tactics execute *actions* that describe the actual adaptation operations. The novelty and important advantage of this modeling approach is that it distinguishes high-level reconfiguration objectives (strategies) from low-level implementation details (reconfiguration tactics and actions) and explicitly separates platform specific adaptation operations from system-independent adaptation plans.

Note that the adaptation process model is intended to specify an adaptation process on the model level. However, the actual execution of the process on the model instance, as well as on the real system, is realized as part of a framework that interprets the modeled adaptation process. More details on the adaptation process meta-model and the framework interpreting the process can be found in [12].

B. Adaptation Control Loop

Now that we have introduced DML, we explain how we employ it to realize proactive model-based system adaptation. Similar to MAPE-K [13] or [7], we distinguish four main phases of the adaption process: COLLECT, ANALYZE, DECIDE, and ACT. Central element of the adaptation control loop depicted in Figure 2 is a DML instance. It represents the KNOWLEDGE about the system and its adaptation processes [15].

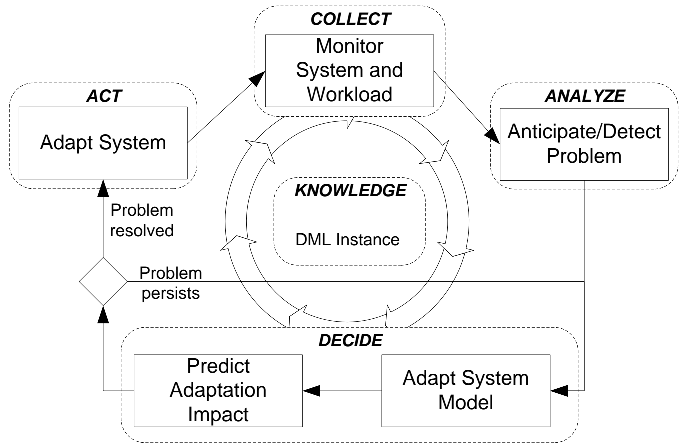

Fig. 2: Model-based adaptation control loop.

In the COLLECT phase, we use monitoring data to extract model instances or to update and calibrate existing instances [4]. During the ANALYZE phase, we use the DML model to support problem analysis, i.e., to reason about the correlation of observed problems and their cause. Online adaptation of software system is useless if the time needed to analyze is too long as compared to the time of changes in the system and the context. The advantage of DML, compared to predictive models (like QN or QPN) is that analysis granularity (black-box, coarse-grained, or fine-grained) and solution technique (e.g., analytical or simulation-based analysis of QPNs or operational laws) can be varied. This enables a tradeoff between solution accuracy and runtime regarding the constraints of a concrete scenario. In the context of the BlueYonder case study, we apply —fully automated— a model-to-model transformation to QPNs [19], a simulation based solution (using SimQPN) and a result back propagation to solve the model. During the DECIDE phase we use the modeled information about the system's adaptation points as well as the adaptation process to adapt the model instance and find a system state that fulfills the system's operational goals. In this phase, we also use the architecture-level performance model for performance analysis to evaluate the impact of the adaptation actions on the system performance. Especially during this phase it is beneficial to have a model that also comprises structural information about the architecture of the system such that it can be used for detailed reasoning about the impact of changes in the system environment as well as the impact of possible adaptation actions.

IV. CASE STUDY

In this section, we apply and evaluate our previously presented approach in the context of our industrial partner Blue Yonder.

A. Motivaiton

Blue Yonder is a leading service provider in the field of predictive analytics and big data. The company offers enterprise software services that are based on predictions of, e.g., sales, costs, churn rates etc. Blue Yonder employs machine learning techniques to obtain accurate predictions based on historical data provided by their customers. Usually supervised machine learning can be applied, consisting of a training step that is used to infer a mathematical model for the available historical data. This model can then be used to calculate forecasts based on a given input data set. Training the model and calculating the forecasts requires a considerable amount of computational resources depending on the amount of customers, their input data, and their service-level agreements (SLAs).

Currently, Blue Yonder uses dedicated resources for each customer to fulfill their respective SLAs. When acquiring new customer projects, Blue Yonder normally has to estimate how much resources are required to sustain the workloads of the new customers and ensure adequate performance. This estimation is based on the experience of Blue Yonder's employees and can range from a few low-budget desktop machines to hundreds of cores on high-end servers, depending on the customer's amount of data to be analyzed and on the time available for the analysis. More importantly, this estimation is generally a worst-case estimation, i.e., the system capacity is dimensioned to support the peak workload intensity.

Given the increasing number of servers and respective operating costs, Blue Yonder is interested in increasing resource efficiency by sharing resources among different customers. As Blue Yonder has detailed information from their customers about when, how many, and which type of requests are expected to arrive (*request schedule*), a self-adaptive approach that assigns the required amount of resources to new customers and dynamically adapts the amount of resources according to the actual customer demand appears to be promising.

Thus, the major goal of this case study is to evaluate if our approach is applicable in Blue Yonder's scenario and if it is capable of increasing resource efficiency. Additionally, it is crucial that the adaptation is capable of considering the different performance requirements of Blue Yonder's customers. The research question targeted in this scenario is whether our approach is applicable in an environment with heterogeneous resources—low-cost desktop computers and high-end servers—and whether it can be effectively used to trade-off different performance requirements of multiple customers.

In the following, we first explain the architecture of Blue Yonder's system and present the respective architecture-level performance model, modeled with the Descartes Modeling Language (DML). Next, we specify an adaptation process using DML's adaptation process model to adapt the system to changes in the environment considering the previous requirements. Finally, we present the results when applying our model-based self-adaptive performance and resource management approach in Blue Yonder's system.

B. Blue Yonder System Architecture

A typical Blue Yonder system consists of three main software component types: the Gateway Server (`GW`), the Prediction Server (`PS`), and a third party component, the database (`DB`) (see Figure 3). The `GW` is the communication endpoint to the Blue Yonder system. Users can invoke a set of different services via HTTP. In the considered sample project, the available services are `train`, `predict` and `results`. As their names suggest, the `train` service initiates the training step of the supervised learning algorithms. The `predict` service initiates the calculation of the forecasts using the trained prediction model. The `results` service makes the final results available to the customer. To train the prediction model, the `train` service accepts historical data. The `GW` receives this data, parses it and generates a job, which is put into the `GW`'s queue and scheduled for processing. Then, an active `PS` takes the job from the queue, processes it (i.e., trains the prediction model) and stores the results in the database. After training, a user can invoke the `predict` service to calculate a forecast based on the trained prediction model. The user sends the data for which the forecast should be made to the `GW`. The `GW` reads the data and generates one or several jobs—depending on the size of the data—which are put into its queue. These jobs are again processed by one or several `PS` and the results are stored in the database for retrieval by the user (`results` service). Technically, `GW` and `PS` are independent operating system processes that can be started and stopped on any machine in the resource landscape. The database for our case study is a standard MySQL database. Each customer has its own `GW`, `PS`, and `DB` instances, which are deployed in Blue Yonder's resource landscape. The number of component instances and their distribution in the system environment is called *topology*.

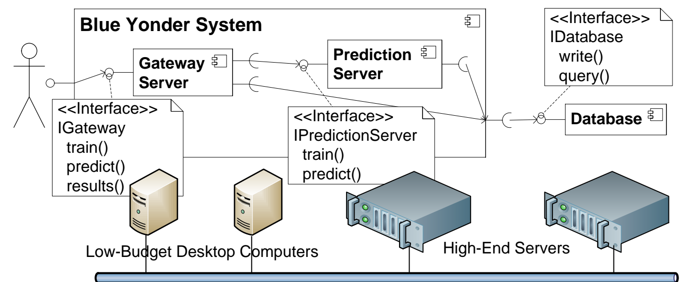

Fig. 3: Example topology of a Blue Yonder system with heterogeneous hardware.

In the scenario evaluated at Blue Yonder, the resource landscape consists of a heterogeneous hardware environment comprising two low-budget dual-core machines (`desc1` and `desc2`) and two high-end quad-core machines with hyper-threading (`desc3` and `desc4`). The example topology is depicted in Figure 3. In the depicted default setup, the database runs on a dedicated high-end machine. `GW` and `PS` instances can be distributed over the two low-budget machines and the second high-end machine. All machines are connected with a

1 GBit Ethernet.

The usually experienced workload in the system can be characterized by the service that is called (`train`, `predict`), the execution type of the requests (sequential or in parallel), and the requests' size (the number of records in the request, typically varying between 10,000 and 500,000). To react on changes in the environment (changes in workloads of existing customers, launching of new customer projects), additional prediction server instances can be started on other machines. Furthermore, prediction server instances can also be migrated between machines at run-time. The challenge in such a setup is that our approach is now faced with a heterogeneous hardware environment and with different performance requirements of multiple concurrent customers. For example, upon a workload change of a given customer, the adaptation process has to decide whether to start/stop a prediction server on a low-budget or a high-end machine while taking into account the performance requirements and topology of other customers.

C. Architecture-Level Performance Model

For our case study, we conducted multiple experiments to construct an architecture-level performance model of the Blue Yonder system. In this section, we first present the implementation of the model and then discuss its performance prediction accuracy. Even though efforts and knowledge to build the model for this scenario where negligible, future research will target a fully automatized extraction of structure and parametrization based on trace data.

1) Model Structure: An overview of the DML instance we created for the Blue Yonder system is depicted in Figure 4 in a UML-like notation. In summary, the figure depicts the resource landscape, application architecture, usage profile, and deployment of Blue Yonder's system. The model also includes parametric descriptions of the performance behavior of the software components.

In the center of Figure 4, we see the resource landscape model, consisting of a DataCenter BYDC that contains the previously described hardware. The resource configuration specifications of the ComputingInfrastructures are attached as annotations. The ComputingInfrastructure nodes are connected with a 1 GBit Ethernet.

On this resource landscape, we have deployed four different component instances, one `GW`, two `PS`, and one `DB`. This depicted deployment is only one instance of the possible deployment variants of Blue Yonder's system. During an adaptation process, when the model is changed, the model instance may look different, e.g., further `PS` instances might be deployed on other machines.

Each of the depicted software components provides one or more services. For some of these services, we have depicted their resource demanding service effect specification (RD-SEFF). As described in Section III-A2, an RD-SEFF is a fine-grained description of the control flow and the resource demands of the performance behavior of the modeled service. For example, the top of Figure 4 depicts the RD-SEFF of the

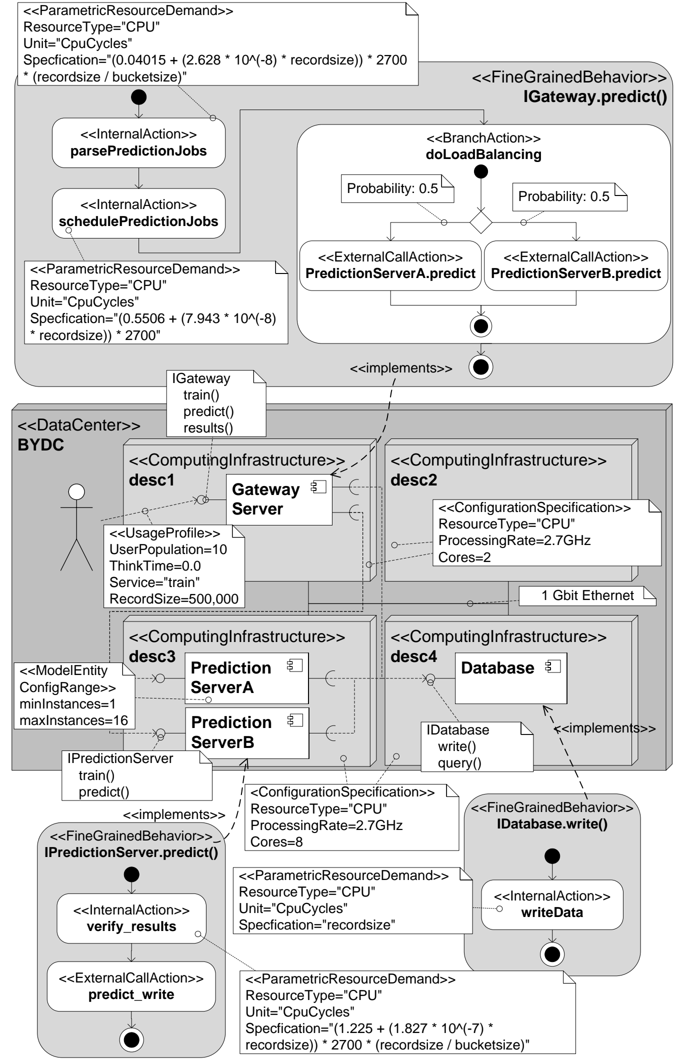

Fig. 4: DML instance describing a deployment of the Blue Yonder system.

`predict` service offered by the `GW` component to the customers. First, it consists of two InternalActions that require a certain amount of CPU resources to parse and schedule the prediction job (parsePredictionJobs and schedulePredictionJobs, respectively). After parsing and scheduling the job, it is passed to one of the available prediction servers. In this example, as there are only two `PS` instances, the probability for each branch is 50%. However, during the adaptation process, when the number of `PS` instances changes, the respective probabilities have to be adjusted accordingly.

Finally, we have to model the usage behavior of the customers that use the system. An example usage profile is depicted on the left where ten customers use the `train` service with a record size of 500,000 in a closed workload scenario with zero thinktime.

2) Parameterizing the Model: The derivation of the resource demands contained in the different RD-SEFFs is im-

portant for accurate performance predictions. To obtain an accurate performance model, we have conducted a large set of experiments varying different parameters to correlate the dependencies of the resource demands on these parameters. The parameters we varied in our experiment are mainly induced by the parameters the users can vary: the called service type (`train` vs. `predict`), the record size (10,000 to 500,000 records per user request), the execution type of the requests (sequential or parallel), and the number of parallel requests (1 to 10). Furthermore, to investigate the impact of the heterogeneous hardware environment and the mutual influences of multiple `PS` instances, we also varied the `PS` deployment (high-end vs. low-budget machine) and the number of `PS` instances (1 to 8).

The metrics we observed during our experiments to derive the resource demands were average CPU utilization and average response time, i.e., the time the user request spends in the system. To obtain these data, we wrote Python scripts that measured the CPU utilization during the experiments using `sar` and extracted the timing values from the log files of the system. We then used the R framework for statistical computing [1] to derive a resource demand using linear regression. An example parametric resource demand of the InternalAction `schedulePredictionJobs` is

$$rd = (0.5506 + (7.943 \cdot 10^{-8} \cdot \texttt{recordsize})) \cdot 2700$$

In this example, the resource demand depends on an external parameter `recordsize` that corresponds to the number of records in the user request. Note that the additional multiplication by 2700 is necessary to adjust the resource demand to the processing rates of the hardware. We have derived such parametric resource demands for all InternalActions in our model.

3) Model Accuracy: To evaluate the accuracy of the performance predictions provided by the model, we conducted several experiments. In Figure 5, we compare the predicted with the measured response times of the `train` and `predict` services for five parallel user requests with a varying amount of `PS` instances. The figure shows that the response time of the `train` service improves when we increase the `PS` instances up to a number of five, whereas the `predict` response time improves further. The reason is that the five parallel `train` requests can be load-balanced to five `PS` instances. In contrast, the `predict` requests can be split into more than five jobs which can be distributed over further `PS` instances to speed-up performance. This confirms that the modeled behavior of the Blue Yonder system is correct.

In another scenario we evaluated the prediction accuracy for different workload mixes. Table I shows the absolute and relative prediction error of the average response time and Table II the absolute prediction errors for the CPU utilization on different hardware nodes.

We also conducted further experiments to evaluate the accuracy in situations where multiple customers use the system

[1] R project: http://www.r-project.org/

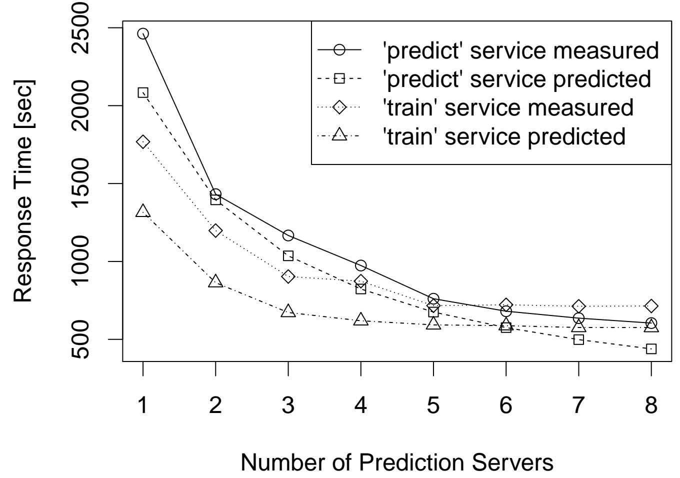

Fig. 5: Comparison of predicted and measured response times for the `train` and `predict` service for five parallel requests and a varying number of `PS` instances running on the high-end machine `desc4`.

TABLE I: Measured and predicted average response times and their relative errors for nine parallel `predict` requests for varying mixed data record sizes with six `PS` instances allocated on `desc4`.

Record Sizes	**Response Time [sec]**		**Error**
[in 1,000 records]	measured	predicted	[in %]
50 & 100	194.50	175.26	-9.9
100 & 200	366.48	325.16	-11.3
150 & 300	545.05	485.65	-10.9
250 & 500	937.24	780.91	-16.7

in parallel or where we vary the amount and deployment of the `PS` instances. All results showed a prediction error in the range of approx. 30%. Due to space limitations, we omit depicting detailed results here. More details about the Blue Yonder model instance, its parameterization, and the prediction accuracy evaluation can be found in a thesis developed in parallel to this paper [23].

D. Adaptation Process

To apply our model-based performance-aware resource management approach, we first have to define and model the adaptation points of the system we want to adapt. Based on the adaptation points, we can then model the adaptation process.

1) Adaptation Points Model: In our case study, we consider the following adaptation points for the Blue Yonder system (see Figure 4 for an example). These adaptation points are customer specific, i.e., they exist for each customer's `PS` instances deployed in the system. First, we can increase or decrease the number of `PS` instances that are assigned to a customer (adaptation points psInstances in Figure 4). The minimal number of `PS` instances is one. The maximum number of `PS` instances is limited by Blue Yonder to two times the available cores, to avoid significant performance degradation due to resource contention. In the adaptation points model,

TABLE II: Measured and predicted average CPU utilizations and their absolute errors for nine parallel `predict` requests for varying mixed data record sizes with six `PS` instances allocated on `desc4`.

Record Sizes [in 1,000 records]	`desc2` [%] meas.	pred.	err.	`desc3` [%] meas.	pred.	err.	`desc4` [%] meas.	pred.	err.
50 & 150	9.42	27.6	18.2	17.11	6.0	11.1	51.56	38.9	12.7
100 & 200	10.35	19.4	9.1	16.54	4.0	12.5	49.19	40.8	8.4
150 & 300	10.51	16.2	5.7	16.33	3.3	13.0	47.79	41.4	6.4
250 & 500	10.20	13.7	3.5	16.65	2.8	13.9	44.67	41.9	2.8

these numbers are specified as OCL constraints that can be checked on the architecture-level performance model.

The second adaptation point is the deployment of `PS` instances. By starting and stopping `PS` instances or by consolidating `PS` instances on fewer machines, we can improve resource efficiency and lower operational costs. For example, it can be beneficial to consolidate the `PS` instances of multiple low-budget machines on a single high-end machine.

2) Adaptation Process Model: Input for and trigger of the adaptation is an updated *request schedule*. As previously mentioned, the request schedule specifies which customers will request what service and with what amount of data. Furthermore, it also contains the customer-specific SLAs. The purpose of the adaptation process presented in the following is to allocate available resources among Blue Yonder's customers such that their SLAs are fulfilled. Moreover, the resources should be used as efficiently as possible to save costs. Thus, the output of our adaptation process is a deployment model that fulfills the required SLAs using resources as efficiently as possible. The found deployment model can then be transformed into a topology configuration that can be used to reconfigure the Blue Yonder system accordingly. In the following, we describe an adaptation process that fulfills these requirements. It is modeled using the adaptation process modeling language presented in Section III-A6. To execute the adaptation process, we use our adaptation framework [**?**] that interprets the adaptation process model instance and adapts the previously introduced architecture-level performance model accordingly.

Essentially, our adaptation process consists of the following strategies: `findDeployment`, `consolidateDeployment`, `reduceDeployment`, and `resolveResourceBottleneck`. The first strategy `findDeployment` allocates new `PS` instances on machines until all customer SLAs are fulfilled. The second strategy `consolidateDeployment` migrates `PS` instances between machines to improve the efficiency. The third strategy `reduceDeployment` removes unnecessary `PS` instances from machines to save operating costs, e.g., if the workload of a customer has decreased. All these strategies exist for each customer. Furthermore, as the process should be able to trade-off the requirements of different customers, we have also defined an additional strategy `resolveResourceBottleneck` that aims at resolving possible resource bottlenecks that may arise when customers share resources. Figure 6 depicts a schematic representation of these strategies for the two different customers A and B.

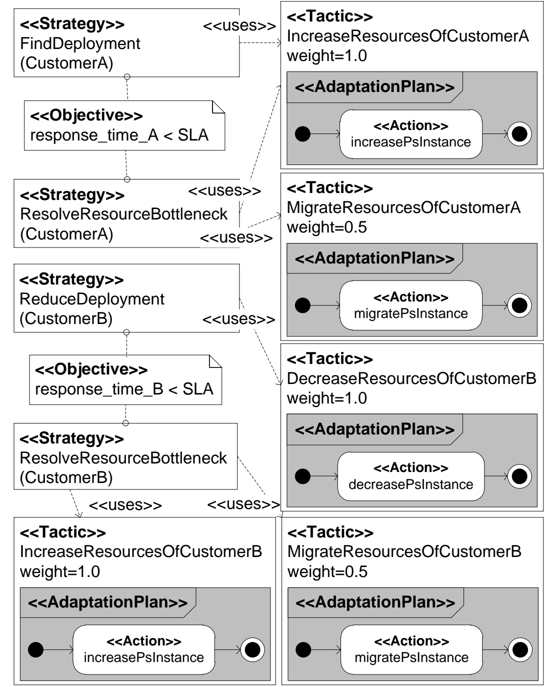

Fig. 6: Schematic representation of the modeled `resolveResourceBottleneck` strategies.

E. Evaluation

In this section, we present the evaluation results of our case study. The focus of our evaluation was to investigate the applicability and effectiveness of our approach. The following scenarios show that with our model-based approach, Blue Yonder is able to adjust the amount of used resources to changes in the customer workloads. Furthermore, we show that the approach is applicable in a heterogeneous resource environment and that it can trade-off diverging performance requirements of different customers.

1) Scenario 1: Adjusting resources to workload changes considering a heterogeneous resource environment: The goal of this scenario is to evaluate the effectiveness of our approach in adapting resource allocations to workload changes such that customer SLAs are fulfilled, considering the heterogeneous hardware resources.

Our scenario starts with the default topology (one `GW` on `desc2`, one `PS` on `desc4`, one `DB` on `desc3`) and a customer that issues one `predict` request with 500,000 records. We assume that all records of this customer must

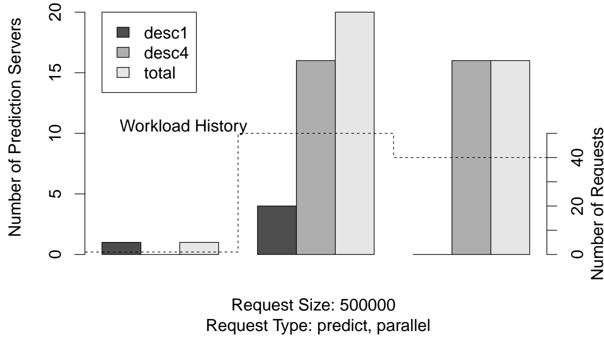

Fig. 7: Adaptation of the system environment to changes in the workload of a customer.

be completed within 3600 seconds (one hour). The default topology is able to handle this load without SLA violations. However, if we increase the number of requests from one to 50, the default topology is not able to handle the load within one hour. This triggers the DECIDE phase in which our modeled adaptation process suggests a deployment of 20 PS, four on desc1 and 16 on desc4. This deployment is suitable to handle the increased load and shows that our approach utilizes the available capacity, but does not exceed Blue Yonder's specified resource limits. Blue Yonder recommends to limit the maximum number of PS instances per machine to two times the available cores to avoid significant performance degradation due to resource contention. For example, for the high-end machines with four cores and hyper-threading (which is comparable to eight logical cores), 16 PS instances can be executed in parallel with negligible resource contention. However, if the algorithm tries to deploy more PS instances on the machine, the performance decreases due to resource contention.

In the next step, we reduce the workload from 50 to 40 parallel requests, i.e, less resources should be sufficient to maintain the SLA of one hour. Figure 7 shows that our approach reduces the number of PS instances in the system. As a result of the adaptation, the four PS instances running on the low-budget machine are released. This demonstrates that our model-based adaptation approach effectively takes into account the properties of the heterogeneous environment. Removing PS instances from the high-end machine makes no sense because after system adaptation, there would still be two active physical machines. However, by releasing resources from the low-budget machine, we can deactivate this machine. Depending on the scenario, the machine can be put into stand-by mode or it can be made available for other customer projects.

2) Scenario 2: Trading-off resource allocations between customers: In this scenario, we show that our approach is applicable in scenarios where changes of the workload behavior of one customer affect the performance experienced by other customers. The goal is to show that our approach, compared to trigger-based approaches, can trade-off different performance requirements of customers with different priorities.

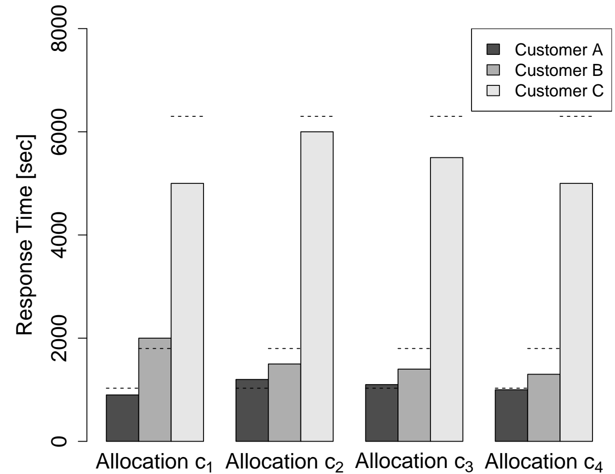

Fig. 8: Response times for the different customers during the adaptation process (SLAs are denoted by the dashed lines).

The initial Blue Yonder topology in this scenario comprises four PS instances that are deployed on desc1 (see Figure 9). Two of theses PS instances belong to customer A, which is a gold customer. The other two PS instances belong to customer B and C, respectively, which are silver customers. A gold customer, in this scenario, is a customer with higher priority, i.e., violating its SLAs causes higher penalties.

To minimize penalties, PS instances of gold customers must not be executed on machines that are overcommitted, i.e., machines executing more PS instances than are executable in parallel (see previous section). The trigger of the adaptation is an SLA violation of customer B due to an increase in his workload (see Figure 8). Our adaptation process first starts another PS instance for customer B on desc1 to solve this problem (topology c_2). However, in this new topology, the SLAs of the gold customer A are now violated. Therefore, the adaptation process continues and migrates a PS instance of customer A to desc4. The migration reduces response times, but still does not eliminate the SLA violation. Therefore, the adaptation process continues and migrates the second PS instance of customer A to desc4 (topology c_4). This resolves the problem and the adaptation process completes.

This scenario shows how our model-based approach can explicitly take into account the impact of an adaptation on other applications (adding a new PS instance for customer B affected customer A) and can automatically find a way to resolve SLA violations at the model-level. In the considered scenario, a conventional trigger-based approach would simply add a PS instance. The problem that adding this further instance leads to an SLA violation of customer A, would be detected after the system has been adapted. Of course, then a new trigger would start further adaptations to solve the new problem, but penalty costs would arise due to SLA violations that will most likely have already occurred.

This scenario demonstrates the contrast between reactive

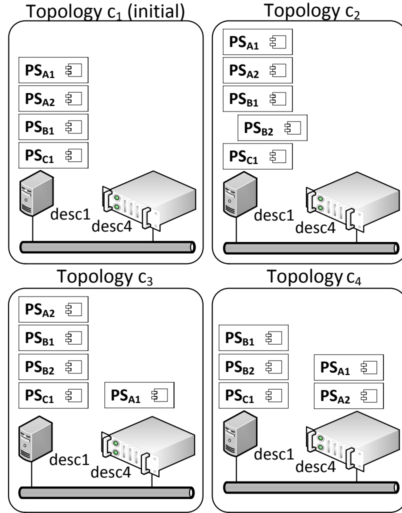

Fig. 9: Details of the different system configurations during the adaptation to a workload change of a single customer affecting other customers.

trigger-based adaptation approaches and our proactive model-based approach that can predict the effect of possible adaptations. By using models we can avoid a "trial-and-error" behavior and guide the adaptation to effectively ensure SLA compliance while maximizing efficiency.

3) Scenario 3: Efficient Resource Usage: In this scenario, we show that the allocation found by the modeled process provides a valid solution that uses resources efficiently. Like in the previous experiments, we use a workload of five parallel `predict` service requests with 500,000 data records. Moreover, the SLA with the customer states that the request has to be processed within 800 seconds. For such a scenario, our approach suggests to allocate five `PS` instances on `desc4` to maintain the SLA.

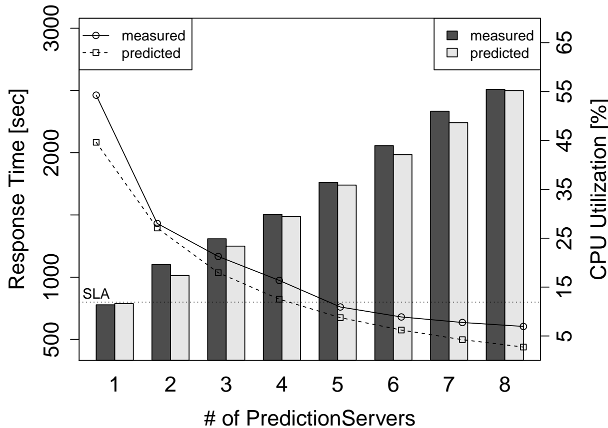

Fig. 10: Comparison of predicted and measured metrics for five parallel `predict` requests with a record size of 500,000. Prediction servers have been started on the high-end machine `desc4`.

To evaluate the quality of this solution, we compare measured and predicted average response time and CPU utilization for this scenario with a varying amount of `PS` instances deployed on a high-end machine (see Figure 10). As we can see, four `PS` instances are not enough to meet our deadline of maximum 800 seconds, since four instances need 974 seconds to process the given workload. This figure also shows that further `PS` instances would speed-up the processing of the requests, but would also lead to a higher resource usage.

V. Conclusions

This paper presents a refinement of the classical MAPE-K adaption approach entirely based on architectural performance models. We apply an exemplary end-to-end application of our holistic adaptation approach to a real-life system with representative settings and provide an experimental evaluation demonstrating its effectiveness and practical applicability. More specifically, we presented the results of a case study we conducted at our industrial partner Blue Yonder. We applied the Descartes Modeling Language (DML) to describe the system architecture and performance behavior of Blue Yonder's predictive big data analytics services infrastructure and evaluated the performance prediction accuracy of the created model instance. Moreover, we used DML to describe an adaptation process leveraging the performance model to adapt Blue Yonder's system to changes in its workload.

This paper shows that our architectural model based adaption process can be effectively used to trade-off different performance requirements of multiple customers in an environment with heterogeneous resources consisting of low-cost desktop computers and high-end servers.

As future work, we want to refine the adaptation process and implement more sophisticated adaptation strategies. Furthermore, we plan to integrate a cost function such that we can allow SLA violations in cases where the benefit of resource savings would compensate incurred penalty. Even though the effort for model creation was negligible, the automation could be further improved towards full automation. Additionally, it would be interesting to evaluate if this case study and its results are reusable in the context of other approaches working on engineering self-adaptive software systems.

References

[1] A. Agrawal, G. Karsai, and F. Shi. A UML-based graph transformation approach for implementing domain-specific model transformations. *Journal on Software and Systems Modeling*, pages 1–19, 2003.

[2] M. Becker, M. Luckey, and S. Becker. Model-driven performance engineering of self-adaptive systems: a survey. In *QoSA*, 2012.

[3] G. Blair, N. Bencomo, and R. France. Models@Run.time. *Computer*, 42(10):22–27, 2009.

[4] F. Brosig, N. Huber, and S. Kounev. Automated Extraction of Architecture-Level Performance Models of Distributed Component-Based Systems. In *Intl. Conf. On Automated Software Engineering*, 2011.

[5] F. Brosig, N. Huber, and S. Kounev. Architecture-Level Software Performance Abstractions for Online Performance Prediction. *Elsevier Science of Computer Programming Journal (SciCo)*, Vol. 90, Part B:71–92, 2014.

[6] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. Engineering self-adaptive systems through feedback loops. In *Software Engineering for Self-Adaptive Systems*. Springer-Verlag, 2009.

[7] B. H. C. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee. Software Engineering for Self-Adaptive Systems: A Research Roadmap. volume 5525 of *Lecture Notes in Computer Science*. 2009.

[8] S.-W. Cheng and D. Garlan. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software*, 85(12):2860 – 2875, 2012.

[9] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. *Theory and Application of Graph Transformations*, pages 296–309, 2000.

[10] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[11] N. Huber, F. Brosig, and S. Kounev. Modeling Dynamic Virtualized Resource Landscapes. In *Intl. Conf. on the Quality of Software Architectures*, 2012.

[12] 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 Journal (SOCA)*, 8(1):73–89, 2014.

[13] IBM Corporation. An architectural blueprint for autonomic computing (White Paper, 4th Ed.), 2006.

[14] G. Jung, M. Hiltunen, K. Joshi, R. Schlichting, and C. Pu. Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In *ICDCS*, 2010.

[15] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[16] S. Kounev, F. Brosig, N. Huber, and R. Reussner. Towards self-aware performance and resource management in modern service-oriented systems. In *IEEE SCC*, 2010.

[17] H. Koziolek. Performance evaluation of component-based software systems: A survey. *Performance Evaluation*, 2009.

[18] J. Li, J. Chinneck, M. Woodside, M. Litoiu, and G. Iszlai. Performance model driven QoS guarantees and optimization in clouds. In *ICSE Workshop on Softw. Eng. Challenges of Cloud Computing*, 2009.

[19] P. Meier, S. Kounev, and H. Koziolek. Automated Transformation of Component-based Software Architecture Models to Queueing Petri Nets. In *Intl. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2011.

[20] M. N. Bennani and D. A. Menascé. Resource Allocation for Autonomic Data Centers using Analytic Performance Models. In *Proc. of the 2nd Intl. Conf. on Automatic Computing* , 2005.

[21] P. Oreizy, M. Gorlick, R. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf. An architecture-based approach to self-adaptive software. *Intelligent Systems and their Applications, IEEE*, 14(3):54–62, 1999.

[22] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):14, 2009.

[23] W. Schott. Automated model-based system reconfiguration: A case study. Master's thesis, KIT, 2013.

[24] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, 2002.

[25] G. Toffetti, A. Gambi, M. Pezzè, and C. Pautasso. Engineering autonomic controllers for virtualized web applications. In *Web Engineering*, volume 6189 of *LNCS*. 2010.

[26] T. Vogel and H. Giese. A language for feedback loops in self-adaptive systems: Executable runtime megamodels. In *SEAMS*, 2012.

[27] T. Vogel, A. Seibel, and H. Giese. The Role of Models and Megamodels at Runtime. In *Models in Software Engineering*, pages 224–238, 2011.

[28] Q. Zhang, L. Cherkasova, and E. Smirni. A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications. In *Intl. Conf. on Autonomic Computing*, 2007.