Pre-print version for personal use only!

Chapter 16
Run-time Models for Online Performance and Resource Management in Data Centers

Simon Spinner, Antonio Filieri, Samuel Kounev, Martina Maggio, and Anders Robertsson

Abstract In this chapter, we introduce run-time models that a system may use for self-aware performance and resource management during operation. We focus on models that have been successfully used at run-time by a system itself or a system controller to reason about resource allocations and performance management in an online setting. The chapter provides an overview of existing classes of run-time models, including statistical regression models, queueing networks, control-theoretical models, and descriptive models. The chapter contributes to the state-of-the-art, by creating a classification scheme, which we use to compare the different run-time model types. The aim of the scheme is to deepen the knowledge about the purpose, assumptions and structure of each model class. We describe in detail two modeling case studies, chosen because they are considered to be representative for a specific class of models. The description shows how these models can be used in a self-aware system for performance and resource management.

Simon Spinner
University of Würzburg, Department of Computer Science, Am Hubland, D-97074 Würzburg, Germany, e-mail: simon.spinner@uni-wuerzburg.de

Antonio Filieri
Imperial College London, Department of Computing, 180 Queen's Gate, SW7 2AZ, London, UK
e-mail: a.filieri@imperial.ac.uk

Samuel Kounev
University of Würzburg, Department of Computer Science, Am Hubland, D-97074 Würzburg, Germany, e-mail: samuel.kounev@uni-wuerzburg.de

Martina Maggio
Lund University, Department of Automatic Control, Ole Römers väg 1, SE 223 63 Lund, Sweden, e-mail: martina.maggio@control.lth.se

Anders Robertsson
Lund University, Department of Automatic Control, Ole Römers väg 1, SE 223 63 Lund, Sweden, e-mail: anders.robertsson@control.lth.se

Pre-print version for personal use only!

16.1 Introduction

Modern data centers provide advanced mechanisms to dynamically change the amount of physical resources (compute, storage and network resources) allocated to deployed applications. The allocation of resources has a strong influence on the performance of an application (measured by the end-to-end response time and throughput) as experienced by its end users. In order to achieve a certain level of performance, e.g., specified in a Service-level Agreement (SLA), the application requires a certain amount of physical resources. However, this resource requirement depends on many performance influencing factors, including the current workload, the application implementation, the configuration of underlying layers (such as middleware systems, operating systems, or virtualization), and external services invoked by the application. As a result, the relationship between resource allocations and application performance is highly *non-linear* for most practical systems and subject to *frequent changes* (e.g., time-varying workloads, dynamic system reconfigurations).

Self-aware systems for online performance and resource management in data centers need to learn models that capture the complex relationship between resource allocations and application performance, taking into account the various performance-influencing factors. Based on these models, self-aware systems will help to increase resource efficiency while ensuring a certain level of application performance in data centers (see also the reference scenario on data center resource management in Chapter 4, as they enable to reason and act based on models capturing knowledge about the system's performance behavior. All factors expected to change over time should be explicit parameters of such models, so that a self-aware system is able to evaluate the impact of changes in these factors on the application performance in advance, and proactively reconfigure itself when necessary to improve resource efficiency or avoid SLA violations.

In this chapter, we give an overview of different classes of models that have been used at system run-time for online performance and resource management and discuss their applicability for building self-aware systems, where self-awareness is considered with respect to the system's performance and resource management behavior. Furthermore, we present two candidate models in more detail showing how these models can be used as abstractions of practical systems supporting allocation decisions in data centers.

In the past, there has been comprehensive work on offline, design-time models for comparing different design alternatives and for offline capacity planning purposes (e.g., queueing models, petri nets). These models are also partially applicable in online scenarios, as we will see later. On the other hand, we see fundamental differences between design-time and run-time models [27]:

1. At design-time, we can use time-intensive, computationally expensive techniques (e.g., discrete-event simulation) to analyze a model. At run-time, the analysis is often time-critical and the analysis overhead is an important factor. Therefore, it is often necessary to decide on a model abstraction level and an analysis technique that provides a trade-off between accuracy and overhead.

Pre-print version for personal use only!

2. Design-time models are usually constructed manually based on experiments in dedicated test environments or traces from a production system covering a limited period of time. Run-time models should be constructed automatically from monitoring data of the production system. The system should be continuously monitored and changes in the system should be automatically reflected in the models.
3. At design-time, it is usually possible to explore a large space with many degrees-of-freedom. In contrast, a run-time model should be specifically targeted at the degrees-of-freedom that can be changed at system run-time in order to limit the search space.
4. Run-time models should be able to reflect the layered architecture of applications in data-centers (virtualization, operating system, application) and support to answer performance-related questions relevant to the different layers (e.g., the data center owner often has different goals from those of the application owner). Design-time models are optimized for their usage in different phases of the software lifecycle (e.g., design, development, or in production).

In this chapter, we focus on the different classes of run-time models and their solution techniques that can be used to analyze such models. Chapter 17 will introduce techniques to automatically extract models from running applications and update these models when the system changes.

16.2 Run-time Models

This section provides an overview of existing classes of run-time models and shortly introduces their main aspects. The different classes of models are compared along the dimensions described in Section 16.2.1. The classes of run-time models are then presented in Sections 16.2.2 to 16.2.6.

16.2.1 Dimensions for Classification

The classification of the run-time models is based on the following five dimensions:

- *Abstraction level:* We distinguish between the following abstraction levels for describing a system: black-box, coarse-grained and fine-grained [27]. A *black-box* model describes the functional relationship between the inputs and outputs of a system without any knowledge of the system internals. A *coarse-grained* model includes information on the internal architecture of a system (e.g., components, resources, control flow between components). A *fine-grained* model may include additional descriptions of the component-internal behavior (e.g., forks and synchronization, dependencies on input parameters). The abstraction level can influence the analysis accuracy and the types of questions that can

Pre-print version for personal use only!

be answered by a model. For example, predicting the influence of deployment changes on the performance requires a representation of the internal system architecture in the model.

- *Model structure:* This describes the main elements of the modeling formalism. In particular, we discuss which of these elements can be automatically determined at run-time (i.e., *online*) and which of them are specified manually at design time (i.e., *offline*).
- *Input and output parameters:* A performance model can be seen as a generic function $\mathbf{y} = f(\mathbf{x})$ that calculates a vector of output parameters $\mathbf{y}$ from a vector of input parameters $\mathbf{x}$. The output parameters are typically certain performance metrics of the system (e.g., response time, throughput, or resource utilization). The input parameters determine the types of questions that can be answered by a performance model. Typical input parameters are, for instance, workload intensity or number of replicated server instances. In case of dynamic models, where the function also depends on previous outputs, f is a transfer function that maps the input signal x (time-based or frequency-based) to an output signal y. The function itself may not have a closed-form solution.
- *Model inference:* When learning models at run-time, certain techniques are required to determine the model structure and the values of internal model parameters (e.g., service demands of queueing networks). We list state-of-the-art techniques for model inference here and refer the reader to Chapter 17 for details.
- *Model analysis:* Depending on the class of models different techniques for model analysis are available. We distinguish between: *analytical solution* (exact or approximate) and *simulation*. Given that the available time for model analysis is often limited at run-time, fast closed-form or analytical solutions are preferable. However, in certain situations simulations may also be feasible.

16.2.2 Regression Models

Regression techniques can be used both for model inference (e.g., for model identification or model parameter estimation), as well as for extracting a regression model that can be applied directly for performance prediction (e.g., extra- and interpolating measured data). In this section, we focus on the latter.

Abstraction level: Regression models are mainly used for black-box modeling of a system, where no information (or very limited information) on the internal behavior and the system structure is available.

Model structure: Different types of regression models are available. Linear models are often insufficient to describe the performance of a system (e.g., the function between the incoming workload and the response time of a system is highly non-linear). Non-parametric regression approaches (e.g., Multivariate Adaptive Regression Splines (MARS) [17], Classification and Regression Trees (CART) [8], M5

Pre-print version for personal use only!

Trees [39], Cubist Forests [29]) do not require a decision on the model function to be made a priori (linear vs. polynomial vs. exponential).

Input and output parameters: Regression models directly fit a function describing the relationship between one or several output parameters and multiple input parameters. In theory, all measurable quantities can be used as input or output parameters. Common candidates for output parameters in the context of performance and resource management are response time, throughput, resource utilization, or power consumption. Input parameters could be, for instance, workload intensity, resource allocation, or resource utilization.

Model inference: Linear regression models are typically learned with least-squares regression or robust regression schemes. Non-parametric regression models require specialized model inference algorithms [8, 17, 29, 39].

Examples: Curtois and Woodside use regression splines to derive resource functions for the CPU demand of TCP/IP communication [12]. Noorshams et al. [35] use a similar approach to create black-box models of storage systems.

16.2.3 Queueing Network Models

Queueing networks (QNs) constitute a classical formalism for the performance analysis of computer systems. A QN consists of a set of interconnected queues. Each queue consists of a waiting line and one or multiple servers. Incoming jobs to a queue need to wait as long as all servers are occupied by other jobs. Each job consumes a certain amount of service time at a server. While QNs are traditionally used for predicting the performance of a system for offline capacity planning, they also provide a powerful formalism for online performance and resource management.

Abstraction level: QNs are well suited to describe a system as a black-box (i.e., the complete system is represented by a single queue) or on a coarse-grained level (i.e., individual servers or individual resources are modeled as separate queues). While traditional QNs can describe the control flow between queues, they lack the expressiveness to describe more fine-grained behavior, such as, software synchronization aspects or the hierarchical layering of systems. Extended QN formalisms have been proposed to overcome these limitations, e.g., Extended Queueing Networks or Layered Queueing Networks (LQNs). These extended formalisms also support fine-grained models. It is possible to distinguish between different workload classes in order to obtain per-class performance measures.

Model structure: A QN definition consists of the workload description, the queue descriptions and the service demands. The workload description defines a set of workload classes. For each workload class, the type (open vs. closed) and the workload intensity are specified. The workload intensity is either the arrival rate for open workloads or the number of users and their think time for closed workloads. Each queue description consists of a scheduling strategy (e.g., processor-sharing or first-come-first-serve), a maximum capacity, and a number of servers (i.e., the level of parallelism). The service demand D at a queue is defined as $D = V \cdot S$, where V

Pre-print version for personal use only!

is the visit count of jobs of a workload class at the queue and S is the service time consumed during each visit.

When using QNs at run-time, usually a hybrid approach is chosen where some of the model elements are determined offline and others are updated online. In existing appproaches, the workload classes, scheduling strategies, maximum capacities and service time/inter-arrival distributions are mostly set offline at design time. The number of queues and the number of servers at each queue may be derived easily online from structural information about the system. Workload intensities, service demands and visit counts are ideally updated online based on monitoring data because their values often change during system operation.

Input and output parameters: In theory, all parameters of the workload description (e.g., workload intensity and transaction mix), the queue descriptions (e.g., scheduling strategies) and the service demands are potential input parameters. However, when using QNs for performance and resource management at run-time, most of these parameters are kept fixed, and only a small subset is varied at a given point in time. Typical examples for variable input parameters are the workload intensity, which is varied to predict the impact of changes in the workload, or the number of queues, typically varied to predict the effect of a horizontal scaling action. The output parameters of a QN are performance measures for the complete system as well as for individual queues. These measures are response time, throughput, utilization, average number of users in the system, and queue length. Depending on the employed analysis technique, the results may include mean values, percentiles or complete distributions of relevant metrics such as response time.

Model inference: A key parameter of QNs is the service demand of a request/job at a queue (also referred to as resource demand). In practical systems, service demands often cannot be observed directly. Therefore, extensive research has been done to estimate service demands using statistical techniques (e.g., least-squares regression, Kalman filters, and optimization techniques) based on indirect measurements (for an overview see the survey by Spinner et al. [41]). Other work uses Independent Component Analysis to automatically group requests into workload classes based on their service demands [40]. Complete frameworks for model inference have been proposed for QNs in [31] and for LQNs in [21].

Model analysis: A broad set of techniques for solving QNs are available with different degrees of accuracy and computational complexity. Operational analysis and bounds analysis provide closed-form equations to quickly calculate average performance measures of individual queues [5]. Mean Value Analysis [5] can be used to obtain accurate or approximate average measures for QNs with closed workloads. Assuming certain distributions and scheduling strategies, analytical solvers based on Markov chain analysis [5] can provide fast and accurate results (including percentile measures). However, they often suffer from a state space explosion. Fluid analysis [6] can overcome these limitations, however, it is only an approximative technique. With discrete-event simulation it is possible to analyze any type of QN, however, its computational complexity is prohibitive in many run-time scenarios.

Examples: Menascé et al. [32] propose an SLA controller based on a QN to optimize a system configuration periodically. Chen et al., [11] use QNs to optimize

Pre-print version for personal use only!

the allocation of applications to servers. Bennani and Menascé [4] use the results of an analytically solved QN to improve the deployment of applications in data centers. Pacifici et al. use QNs for performance management of cluster web services [38]. Zhang et al. [46] model multi-tier applications with QNs for resource allocation purposes. Urgaonkar et al. [43] also use QNs and predict the response times of a multi-tier application at run-time using approximative mean value analysis. Li et al. [30] use LQNs for the run-time management of cloud applications. Mistral [23] is a resource management framework for virtualized environments that is based on LQNs.

16.2.4 Petri-Net Models

Petri nets are a mathematical formalism to describe the behavior of distributed and parallel systems. A basic Petri net consists of a set of places, transitions and tokens. Each transition is connected to a number of input places and a number of output places. A transition requires a certain number of tokens in each input place. When this condition is met, a transition is enabled and may fire. When firing, the transition consumes the required tokens and produces new ones in the output places. The firing order of enabled transitions is by default non-deterministic.

In order to capture timing aspects of systems, Stochastic Petri nets have been proposed introducing transitions with a probabilistic firing delay. In this section, we focus on Queueing Petri nets (QPNs) [2] as an extension of the Stochastic Petri nets including so-called queueing places to model scheduling aspects.

Abstraction level: QPNs support black-box, coarse-grained and fine-grained models. QPNs can be used to model the fine-grained control flow within individual components including synchronization aspects in parallel systems. QPNs also support different token colors to distinguish between different types of requests.

Model structure: A QPN is an 8-tuple where (P,T,C,I^-,I^+,M_0,Q,W) [2]. P is a set of places and T a set of transitions. The color function C assigns a set of supported colors to each place and each transition. I^- and I^+ are the backward and forward incidence functions defining the firing of transitions. M_0 is the initial marking of tokens to places. Q contains a queue description similar to that of QNs for each queueing place. W assigns firing weights to each transition. Existing uses of QPNs at run-time are focusing on online updates of the service time parameters of queueing places (contained in Q) and the initial marking of the QPN (e.g., to specify the number of concurrent users). In [36], also the structure of the QPN is updated at run-time as well to reflect changes in the system architecture (horizontal scaling). Based on an initial offline model all parameters of the QPN are updated online in this case.

Input and output parameters: All elements of the QPN 8-tuple can be considered as input parameters. However, typical questions for online performance and resource management (e.g., horizontal scaling of servers) require adaptations to only a few of those parameters. Therefore, it may be beneficial to define specific input

Pre-print version for personal use only!

parameters for the available reconfiguration options (e.g., number of replicated instances) of a system and define a mapping to the respective QPN elements. QPNs support the same output parameters as QNs on a system of a per-place level, namely response time, throughput, utilization, number of users in the system, and queue length. In addition, it is also possible to determine the usage of software resources (e.g., thread or connection pools).

Model inference: The same techniques used for QNs to estimate service demands can be applied to QPNs.

Model analysis: While QPNs support a range of qualitative analyses, these are irrelevant for online performance and resource management. Quantitative analysis can be done based on Markov Chain analysis [2]. However, for many practical applications one has to resort to simulation [42] due to a state space explosion.

Examples: In [36], QPNs are used to schedule jobs in the context of the Globus Grid computing framework such that performance requirements are satisfied.

16.2.5 Control-theoretical Models

Control theory offers a principled approach and a set of guidelines for how to create a control system. Its applications spread in many engineering domains most of which interact with the physical world, such as an industrial plant and chemical processes. In general terms, a *controller* takes inputs form the *sensors* that measure the environment, determines a sequence of actions that drive the plant toward its intended behavior, and executes these actions through the *actuators*. The major advantage of using control theory is the access to a broad set of mathematically grounded techniques that can provide formal guarantees on the effectiveness and robustness of the controlled system [14, 16]. Examples of such guarantees are stability (i.e., the ability of reaching an area close to the desired goal and not leaving that area if the conditions are unchanged), quantified settling time, and the absence of overshoots (possibly corresponding to costly overprovisioning).

Despite the potential benefits of control theoretical solutions for software adaptation and resource management, their usage in practice is challenged by the difficulty of abstracting software behaviors through convenient mathematical formalisms [16]. This challenge is twofold: on the one hand, classic software design models are often inadequate to capture the relevant time dynamics for control; on the other hand, control theory was originally developed around the physical world, whose dynamics are subject to the laws of physics, while software behavior can be arbitrarily more complex in general.

Abstraction level: Finding the right level of abstraction and identifying the proper set of sensors and actuators is in general challenging [16]. A number of adhoc solutions for specific problems or architectures have been proposed in literature (see related work surveyed by Filieri et al. [16]), as well as some attempts towards the automatic synthesis of controllers [14, 15].

In the following, we briefly review the two most common classes of models enabling the application of control theoretical techniques: Dynamical and hybrid system models.

16.2.5.1 Dynamical System Models

Dynamical system models consist of equations that describe the dynamic behavior of objects in a system. Objects are assumed to have input variables and output variables and a certain internal behavior.

Model structure: A dynamical system can be modeled in two different ways, depending on the physical nature of the model. Continuous-time models are represented in the form of Ordinary Differential Equations (ODEs), while Discrete-time models in the form of Difference Equations. In the control of computing entities, we are usually interested in discrete-time models (since in a computer there is usually no continuous-time physics).

Input and output parameters: A discrete-time dynamical system is described by the equations

$$\begin{cases} x(k+1) = f\left(x(k), u(k), d_x(k)\right) \\ y(k) \quad\quad = g\left(x(k), u(k), d_y(k)\right) \end{cases} \tag{16.1}$$

where $x \in \Re^{n_x}$, $u \in \Re^{n_u}$ and $y \in \Re^{n_y}$ are referred to as the state, input and output vectors respectively, $d_x \in \Re^{n_x}$ and $d_y \in \Re^{n_y}$ the state and output disturbance vectors respectively. The two functions $f(\cdot,\cdot,\cdot)$ and $g(\cdot,\cdot,\cdot)$ are real-valued vector functions of proper dimensions. The number k is an integer index counting the time instants – not necessarily evenly spaced in time. In a more general form, $f(\cdot,\cdot,\cdot)$ and $g(\cdot,\cdot,\cdot)$ could depend on an arbitrary number of real-valued parameters, possibly time-varying. The term "step k" denotes the time span between the k-th and the $(k+1)$-th instants.

The first equation in (16.1) is called the *state equation*, and dictates what the system state will be in the end of step k given what it is at the beginning, and what happens to the system input. The system inputs are assumed to be captured by the values of u and d_x at the beginning of the time step. The state equation represents the dynamic system's character as *difference* equations, i.e., owing to the contextual presence of two subsequent index values. In other words, the state equation gives the system "memory of the past", and explains why the same action generally yields different effects depending on the system condition when it is applied. The input vector u represents *manipulated variables* that can be used to influence the system's behavior, while the state disturbance d_x accounts for any input other than u, i.e., for any external entity that actually influences the system state, and that in some cases can possibly be measured, but never manipulated.

The second equation in (16.1) is called the *output equation*. It is not dynamic, as shown by the presence of a single index value k, while the state equation highlights the relationship between what the values at time $k+1$ and the values at time k are. In most problems of interest, the output equation describes what one measures (vector

Pre-print version for personal use only!

y) to represent the system's behavior. The disturbance vector d_y represents possible alterations of the measurements, e.g., due to noise, but *not* of the actual evolution of the system state.

Model inference: Dynamical system models can either be derived from first principles, yielding so called white-box models, where all parameters are given by already known constants, or models where there is a need to identify or estimate (possibly online) parameters and structure. Grey-box models are those that contain structural information like, e.g., known nonlinearities such as saturation elements, and have parameters that need to be identified empirically to make the model complete, whereas black-box-models do not have any a priori structure imposed. Several different system identification methods in both time and frequency domain can be used; either indirectly via so-called nonparametric identification methods like transient response, correlation methods, and spectral analysis, or directly with parameter estimation methods from linear regression and time-series analysis, like (N)ARMAX, instrument variable and prediction error (PE) methods [22]. From input-output data mixed deterministic-stochastic systems can be found, capturing both the (deterministic) process model dynamics as such and the influence of e.g., (stochastic) disturbances. Linear system models can be represented in several different but equivalent forms, like e.g., characterized by their impulse or step responses, by their transfer function or written in a state-space model form. For the latter, efficient numerical algorithms for subspace state-space system identification (so called N4SID-algorithms) have been derived [20, 45]. The well-known Kalman / Kalman-Bucy filtering [24] can be used for both state estimation and parameter estimation and several of the parameter estimation methods mentioned above can be formulated in recursive algorithms, allowing online estimation of model parameters, for instance, in adaptive control algorithms and observers.

Model analysis: In synthesis, dynamical system models are descriptive. They can be used for reasoning (in the sense that you can use them to build an understanding of the current situation and to predict what is going to happen in the future). The abstraction level can (theoretically) be chosen arbitrarily, *but* the models should describe phenomena that can be quantified, which means that for example it might not be possible to detail the behavior of the system at a specific granularity level because it is impossible to find the correct equations for it. In general, higher abstraction levels are easy to solve while for more detailed ones, the number of state variables increases significantly. The mathematical foundations including differential and difference equations, control theory, identification theory and also machine learning could be used to fine-tune the model parameters. These models are quantitative, in the sense that they only describe things that can be turned into numbers, but they cannot for example describe the difference in two software architectures. They are mainly used to: (a) build an understanding of the systems (for example, using system identification can give insight on how the system behaves), (b) control the system to ensure that it has the desired behavior. The models are mainly online ones, but the structure can be prescribed offline (for example, determining that there is a linear relationship between the value of the states at time $k+1$ and the same

Pre-print version for personal use only!

Pre-print version for personal use only!

values at time k). Coefficients can be identified online, or the entire structure itself can be identified online.

16.2.5.2 Hybrid System Models

Model structure: In contrast to the pure discrete-event systems of the types described in Sections 16.2.3–16.2.4, where all evolutions of states are caused by discrete events in time, event-based systems in general may be formulated within the hybrid automaton framework [1, 19]. The latter supports both discrete events, for instantaneous transitions between a set of steps, and time-driven dynamics, comprising continuous- and/or discrete-time dynamics, evolving within each step. As special cases, they can therefore describe the models in previous chapters where either the evolution within each step is trivial/non-changing or the system can be satisfactorily modeled and described as difference or differential equations without any transitions to other steps.

This modeling framework can be used to model a large class of systems and has appeared in slightly different forms in various domains, however, often with different foci. In automatic control the focus is often on the continuous behavior, whereas in computer science the emphasis often concerns the discrete aspects.

By being able to accurately capture system behavior comprising both sequencing and events, like turning on and off computers in large server farms, and the dynamic evolution, like approximate fluid models at high load, within the same framework, this model class has large potential to both describe and predict system behavior relevant for resource allocation and accurate performance monitoring.

However, in general, it is very hard to find analytical solutions to a hybrid system consisting of mixed discrete events and continuous dynamics. A restricted (but still powerful and useful) subclass is linear piecewise hybrid systems [7, 33]. As a note of warning, using hybrid automata for modeling a system may easily introduce undesired or false behavior, even if it may seem reasonable from a physical point of view. After a transition to a new step (caused, e.g., by an event triggering a guard function or governed by high-level sequencing), the dynamics of the new active step may be re-initialized. This switching may thus cause a discontinuous right-hand side of an ordinary differential equation which then typically results in lack of existence or lack of uniqueness of solutions [25].

Input and output parameters: Usually, parameters are the transition probabilities from one state to another or follow a deterministic scheduling based, e.g., on timing.

Model inference: Due to the higher complexity these issues may be difficult to discover in the modeling phase. However, hybrid systems are sometimes intentionally modeled without uniqueness to incorporate uncertainty.

Model analysis: As shown in a simple example from [18], a hybrid model of a bouncing ball, where each contact between the ball and the ground cause an impulse according to the Newtonian law of change of momentum, will create an infinite number of mode switches in finite time, called Zeno behavior. It is therefore

Pre-print version for personal use only!

important to incorporate inherent delays in, e.g., context switching to avoid similar, but erroneous analysis as above, caused by too idealistic scenarios in the modeling.

16.2.6 Descriptive Meta-Models

The previously described model classes are based on rigorous mathematical foundations, whereas the focus of this class of models lies in greater model expressiveness and interpretability. Two representatives of this class of models are SLAstic [44] and the Descartes Modeling Language (DML) [28].

Abstraction level: Descriptive modeling languages may be defined at different abstraction levels allowing to model a system at a black-box, coarse-grained or fine-grained level of detail. They may also support hybrid models with mixed abstraction levels (e.g., DML [28]).

Model structure: The model structure is described in a meta-model. The meta-model describes the available model elements (i.e., classes and their attributes) and the relationships between them (i.e., containment and association). Meta-models based on OMG's Meta-Object Facility (MOF) standard [37] are most commonly used (e.g., UML is an example of a modeling language defined using a MOF-based meta-model). Compared to mathematical formalisms (e.g., QNs or QPNs), descriptive meta-models typically introduce a more extensive set of different model elements with richer semantics. The model elements should be direct abstractions of real-world hardware or software entities (e.g., servers, software components, program loops, branches, etc.) in order to support the understandability of the model.

The improved model interpretability is beneficial in both offline and online scenarios: There is a large number of tools for creating graphical and textual editors for MOF-based modeling languages, simplifying the creation of a model of a system and lowering the barrier for less experienced users. In the online case, the additional meta-information provided through different model elements helps to create richer visualization of the learned model enabling self-expressiveness of the system. Furthermore, meta-models allow to enforce additional constraints on the structure of the models avoiding errors when creating models manually or programmatically.

Input and output parameters: The input parameters are usually not defined by the employed meta-model itself, but are rather highly dependent on the used model analysis method. The meta-model may provide high-level goal descriptions from which low-level input parameters are derived (e.g., DML comes with a special query language [28]). The available output parameters also depend on the model analysis method.

Model inference: It is possible to extract meta-model based performance models from monitoring data. As shown in [10] even very fine-grained architecture-level performance models can be extracted using practical monitoring tools with an acceptable overhead. However, this requires fine-grained instrumentation capabilities in the system.

Model analysis: The model analysis is typically supported either through *model-to-model transformations* into one of the previous mathematical formalisms (e.g., QNs or QPNs) or by direct simulation. The benefit of using a descriptive modeling language is the flexibility to use different mathematical solution techniques depending on the analysis goal without the need to maintain different models.

16.3 Modeling Case Studies

16.3.1 Control-theoretical Models

In this subsection, we present a control theoretical model that captures the maximum response time of a cloud application. The model is primitive, yet useful, and has been applied to develop control strategies to keep the maximum response time bounded [26].

Cloud applications serve multiple users through the Internet. Their computations are generally separated into independent and stateless user requests processed by the system [13]. An essential requirement of these applications is that requests should be processed in a time-sensitive way, otherwise unsatisfied users may abandon the service [34]. Therefore, having a model for the maximum time consumed to generate a response is quite useful.

We assume that the maximum response time of a web application, measured at regular time intervals, follows the equation:

$$\begin{cases} x(k+1) = \alpha(k) \cdot u(k) + \delta t(k) \\ y(k) \quad\;\; = x(k) \end{cases} \tag{16.2}$$

i.e., the maximum response time $y(k)$ of all the requests that are served between time index k and time instant $k+1$ depends on a time varying unknown parameter $\alpha(k)$ and can have some disturbance $\delta t(k)$ that is a priori unmeasurable. $\alpha(k)$ takes into account all the variations that can happen in a controlled way, $u(k)$. These include for example booting of a new machine or a resource upgrade. In Equation (16.2), $\delta t(k)$ is an additive correction term that models variations that do not depend on something that can be changed by the user. These include, for example, variation in retrieval time of data due to cache hit or miss. If $\alpha(k)$ is considered as a time-invariant parameter α, and its time-dependency is not present, the state equation of the proposed model is linear and so is the output equation. Otherwise, the equation can be seen as a Linear Parameter Varying (LPV) one and treated accordingly.

This model is quite trivial, but it captures the application behavior enough for a control action to be useful [26]. A controller should aim at canceling the disturbance $\delta t(k)$ and selecting the value of $u(k)$ so that the maximum response time would be equal to the desired value.

Pre-print version for personal use only!

Pre-print version for personal use only!

16.3.2 Descartes Modeling Language

The Descartes Modeling Language (DML) [28] is a descriptive modeling language for online performance and resource management formalized by a set of meta-models based on OMG's Meta Object Facility (MOF). The goal of the modeling language is to provide common abstractions for describing performance and resource management related aspects of modern dynamic IT systems, infrastructures and services. DML was designed from the beginning to support the design of systems exhibiting self-awareness with respect to performance and resource management aspects. As such, the formalism is intended to build reflective models (see Chapter 6) capturing knowledge about the static structure and dynamic behavior of a self-aware system. In the following, we focus on this part of DML. On top of that, DML also provides means to model the high-level goals specified in Service-Level Agreements (SLAs) and the processes to derive intermediate and low-level goals from the high-level ones (for details see Chapter 17).

16.3.2.1 BlueYonder System

Figure 17.4 depicts an excerpt of a DML model in a UML-like notation. The model was created in the context of an industrial case study in cooperation with Blue Yonder GmbH & Co. KG, a leading service provider in the field of predictive analytics and big data. The modeled system (called Blue Yonder system) provides forecasting services used by customers for predicting, e.g., sales, costs, churn rates, etc. These services are based on compute-intensive machine-learning techniques and subject to customer SLAs. In this case study, the DML models were used to predict the resource requirements for a given usage scenario and optimize the resource allocation to reduce Blue Yonder's operating costs.

Application Architecture: The application architecture of the adapted system is modeled after the principles of component-based software systems. A software component is defined as a unit of composition with explicitly defined provided and required interfaces. For convenience, we also use the term *service* to refer to a signature of a software component's interface. A typical Blue Yonder system (depicted in Figure 17.4) consists of three main software component types: the Gateway Server (`GW`), the Prediction Server (`PS`), and a third party component, the database (`DB`). The `GW` is the communication endpoint to the Blue Yonder system. Users can invoke a set of different services via HTTP (`train`, `predict`, and `results`). The `GW` receives historical data for training a prediction model, 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 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 scheduled

Pre-print version for personal use only!

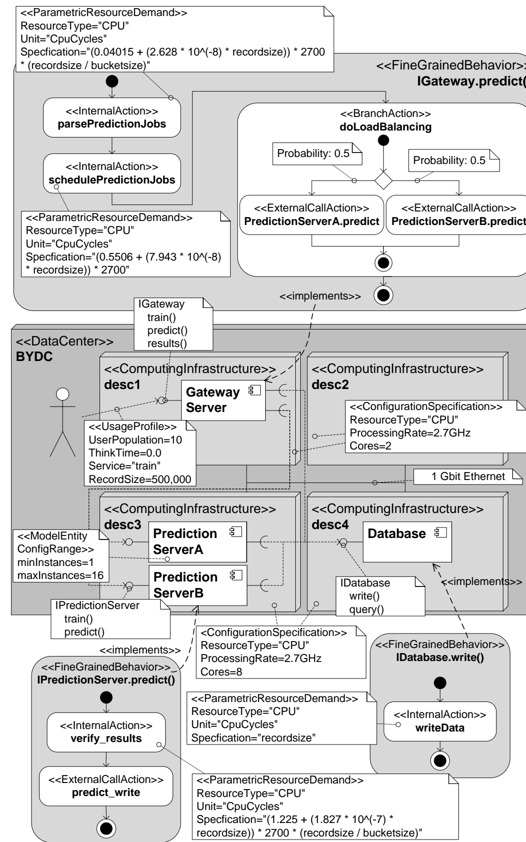

Fig. 16.1: Resource landscape, application architecture, deployment and adaptation points of the Blue Yonder system.

for processing. 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).

The control flow within the components is modeled in Figure 17.4 using a fine-grained behavior abstraction level specifying the sequence of performance-relevant actions (e.g., internal actions, external calls, branches, loops, etc.). For instance, the `predict` service of the `GW` first executes two internal actions (`parsePredictionJobs` and `schedulePredictionJobs`) requiring a certain amount of CPU cycles, which depends on the record size. The record size is an input parameter of the system. The load-balancing behavior is modeled using a probabilistic branch with external call actions to the corresponding `PS` instances. DML supports multiple (possibly co-existing) behavior abstraction levels: fine-grained, coarse-grained (i.e., the service behavior as observed at the component boundaries) and black-box (i.e., a probabilistic representation of the service response time behavior).

Pre-print version for personal use only!

Resource Landscape: The resource landscape meta-model is used to describe the structure and properties of the environment shared by the deployed applications. It includes a description of both the physical and the logical resources of the system environment. The example resource landscape in Figure 17.4 consists of a data center with four different servers (represented by computing infrastructure). Each computing infrastructure is associated with a configuration specification describing the available resources. As the system is compute-intensive, only CPUs are modeled. However, it is also possible to specify other types of resources, such as storage, network and software resources. Additionally, the container hierarchy (e.g., hypervisor, virtual machine, middleware services) including configuration information (e.g., memory, bandwidth, and so on) may be specified.

Deployment: To capture the interactions of the resource landscape and the application architecture, one must model the connection between hardware and software. Each component instance in the system is assigned to a computing infrastructure instance.

Usage Profile: To model user interactions with the system (i.e., the usage profile), DML provides a usage profile meta-model. The usage profile of the `train` service describes a closed workload specifying a user population and think time. In addition to this, the usage profile also specifies the sequence of system calls and values for system parameters used in the system (e.g., record size).

Adaptation Points: The adaptation points model marks the elements of the resource landscape and the application architecture that can be adapted (i.e., reconfigured) at run-time. Adaptation points can be either associated with model parameters (e.g., number of CPU cores) or with model entities (e.g., a software component). In the example model, the adaptation point ModelEntityConfigRange is associated with a component specifying the minimum and maximum number of instances of this component that are allowed. In the example, the `PS` component is instantiated twice (`PredictionServerA` and `PredictionServerB`).

16.3.2.2 Application in Self-Aware Systems

We now discuss *learning* and *reasoning* processes can be implemented using DML.

Learning: We distinguish between *model structure extraction* (e.g., components, interfaces, or control flow) and *model parameterization* (e.g., resource demands, or branching probabilities). The model structure may be extracted using techniques described in Chapter 17. All model parameters (e.g., branching probabilities, resource demands) can be flagged as either *explicit* or *empirical* in DML. Empirical model parameters need to be learned based on monitoring data while explicit ones are specified in advance. Thus, it is possible to specify some model parameters based on expertise knowledge in advance while others are learned at system runtime.

Reasoning: DML can be used to predict the impact of changes in the workload or in the system configuration on the performance and the resource usage of the application. By intention, DML is a purely descriptive model optimized for high expressiveness and good understandability. In order to enable performance predic-

Pre-print version for personal use only!

tions, DML relies on mathematical analysis techniques based on existing stochastic modeling formalisms, such as (Layered) Queueing Networks (LQNs), and Queueing Petri Nets (QPNs). In [9], three different model-to-model transformations are defined providing different levels of prediction speed and accuracy: (a) bounds analysis uses operational analysis from queueing theory to determine asymptotic bounds on the average throughput and response time, (b) an LQN solver enabling the fast analytical solution of models, and (c) a QPN solver that allows to obtain predictions based on simulation. The benefit of the transformation approach is that it provides the flexibility to switch between different prediction techniques depending on the prediction goals. The prediction goals comprise the requested performance metrics, the required accuracy and time constraints.

16.4 Conclusions

16.4.1 Open Challenges

While the described classes of run-time models already provide a foundation for self-aware performance and resource management, we still see several open challenges requiring further research:

- The analysis of models at run-time in a self-aware system is often subject to hard constraints on the analysis time. If the solution from a model is not available in time, it may be too late for a system to react to changes in its environment. Furthermore, the decision-making may require the exploration of multiple alternatives requiring multiple analyses of a model. Therefore, there is a need for fast and flexible analysis techniques especially for coarse- and fine-grained models (e.g., queueing network and Petri net models). This requires advances in optimizing existing analysis algorithms (e.g., by pre-computing certain parts of a model before an analysis request) as well as novel approaches to automate the decision, which analysis algorithm to apply in a given prediction scenario.
- A model is always an abstraction of a real system capturing only a subset of factors influencing the performance of the system. Many factors are often not represented explicitly in the model in order to simplify the model solution or because of being difficult to quantify. Therefore, the resulting performance predictions are always subject to a certain level of uncertainty. New techniques to estimate the uncertainty of model solutions are necessary for a system to be able to take uncertainty into account in its decisions.
- In addition to performance goals, a self-aware system may be subject to constraints and goals with respect to other properties (e.g., reliability, security, energy consumption, or costs). Further models of different types are needed to evaluate and predict such properties under varying system workloads and configurations. Thus, more work is necessary to integrate the different types of models with each other and improve the support for trade-off decisions. De-

Pre-print version for personal use only!

scriptive architecture-level models are beneficial here as additional analyses for different quality attributes can be supported using model-to-model transformations. This has been shown to be beneficial for design-time models (e.g., Palladio Component Model [3]).

- Existing approaches to model-based performance and resource management often either assume the availability of a complete model of the system in advance (e.g., created offline during design and implementation of a system) or at least a model template with relevant parameters estimated at run-time. Only few works consider also the inference of the model structure at run-time. Self-aware systems typically run in dynamic, constantly changing environments. In such environments, learning and maintaining a model automatically at run-time is crucial. Furthermore, it is often not safe to assume that a self-aware system has a global view of its environment. The system may need to interact with other systems to obtain the required information for building models of the environment.

16.4.2 Summary

In this chapter, we surveyed existing classes of run-time models for performance and resource management can be used to support the design of systems with self-aware performance and resource management mechanisms. In particular, we considered regression models, stochastic performance models (QNs and QPNs), control-theoretical models and descriptive meta-models. These models vary in their abstraction level, their model structure, their input and output parameters, as well as. supported model inference and analysis techniques. In general, one can distinguish between *predictive models* (regression, stochastic performance models, or control-theoretical models) and *descriptive models*. The prediction models have a rigorous mathematical foundation optimized for fast and efficient model solution. They are typically purpose-build for answering certain pre-defined questions using a certain analysis technique. In contrast, descriptive models are optimized for greater model expressiveness and interpretability. This class of models can help to abstract from the details of model analysis techniques and thus support a greater variety of analysis techniques by using model-to-model transformations into different prediction models. The decision which analysis technique to use can then be done at run-time depending on the high-level goals of a self-aware system.

References

1. R. Alura, C. Courcoubetisb, N. Halbwachsc, T.A. Henzingerd, P.-H. Hod, X. Nicollinc, A. Oliveroc, J. Sifakis, and S. Yovinec. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(6):3–34, Feb 1995. doi:10.1016/0304-3975(94)00202-T.
2. Falko Bause. Queueing petri nets-a formalism for the combined qualitative and quantitative analysis of systems. In *Petri Nets and Performance Models, 1993. Proceedings., 5th Interna-*

Pre-print version for personal use only!

tional Workshop on, pages 14–23. IEEE, 1993.
3. Steffen Becker, Heiko Koziolek, and Ralf Reussner. The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82:3–22, 2009.
4. Mohamed N. Bennani and D. Menascé. Resource allocation for autonomic data centers using analytic performance models. In *ICAC '05: Proceedings of the Second International Conference on Automatic Computing*, pages 229–240, Washington, DC, USA, 2005.
5. Gunter Bolch, Stefan Greiner, Hermann de Meer, and Kishor S Trivedi. *Queueing networks and Markov chains: modeling and performance evaluation with computer science applications*. John Wiley & Sons, 2006.
6. Maury Bramson. A stable queueing network with unstable fluid model. *The Annals of Applied Probability*, 9(3):818–853, 1999.
7. M.S. Branicky. Stability of hybrid systems: state of the art. In *Proceedings of the 36th Conference on Decision and Control*, pages 120–125, San Diego, California USA, December 1997.
8. Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984.
9. Fabian Brosig. *Architecture-Level Software Performance Models for Online Performance Prediction*. PhD thesis, Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, 2014.
10. Fabian Brosig, Nikolaus Huber, and Samuel Kounev. Automated Extraction of Architecture-Level Performance Models of Distributed Component-Based Systems. In *26th IEEE/ACM International Conference On Automated Software Engineering (ASE 2011)*, 2011.
11. Yiyu Chen, Amitayu Das, Wubi Qin, Anand Sivasubramaniam, Qian Wang, and Natarajan Gautam. Managing server energy and operational costs in hosting centers. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, pages 303–314, New York, NY, USA, 2005. ACM.
12. Marc Courtois and Murray Woodside. Using regression splines for software performance analysis. In *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP '00, pages 105–114, New York, NY, USA, 2000. ACM.
13. Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, May 2002.
14. Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In *Proc. of the 36th Intl. Conference on Software Engineering*, pages 299–310, 2014.
15. Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated multi-objective control for self-adaptive software design. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ESEC/FSE 2015. ACM, 2015.
16. Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, Nicolas D'Ippolito, and Ilias et al. Gerostathopoulos. Software engineering meets control theory. In *Proc. of the 10th Intl. Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2015.
17. Jerome H Friedman. Multivariate adaptive regression splines. *The annals of statistics*, pages 1–67, 1991.
18. Sven Hedlund. *Computational Methods for Optimal Control of Hybrid Systems*. PhD thesis, Department of Automatic Control, Lund University, Sweden, May 2003.
19. T.A. Henzinger. The Theory of Hybrid Automata. In *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 278–292, 1996.
20. B. L. Ho and R. E. Kalman. Effective construction of linear state-variable models from input/output functions. *Regelungstechnik*, 14:545–548, 1966.
21. Tauseef A. Israr, Danny H. Lau, Greg Franks, and Murray Woodside. Automatic generation of layered queuing software performance models from commonly available traces. In *Proc. of the 5th Intl. Workshop on Software and Performance*, pages 147–158, 2005.
22. Rolf Johansson. *System Modeling and Identification*. Prentice Hall, Englewood Cliffs, New Jersey, January 1993.
23. Gueyoung Jung, M.A. Hiltunen, K.R. Joshi, R.D. Schlichting, and C. Pu. Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th Intl. Conf. on*, pages 62 –73, 2010.

24. R. Kalman and R. Bucy. New results in linear filtering and prediction theory. *Trans ASME, J. Basic Eng., ser. D*, 83:95–107, 1961.
25. H.K. Khalil. *Nonlinear Systems*. Pearson Education. Prentice Hall, 2002.
26. Cristian Klein, Martina Maggio, Karl-Erik Årzén, and Francisco Hernández-Rodriguez. Brownout: Building more robust cloud applications. In *Proceedings of the 36th International Conference on Software Engineering*, pages 700–711, 2014.
27. Samuel Kounev, Fabian Brosig, and Nikolaus Huber. The Descartes Modeling Language. Technical report, Department of Computer Science, University of Wuerzburg, October 2014.
28. Samuel Kounev, Nikolaus Huber, Fabian Brosig, and Xiaoyun Zhu. Model-Based Approach to Designing Self-Aware IT Systems and Infrastructures. *IEEE Computer Magazine*, 2016. Accepted for Publication.
29. M. Kuhn, S. Witson, C. Keefer, and N. Coulter. Cubist Models for Regression. `http://cran.r-project.org/web/packages/Cubist/vignettes/cubist.pdf`. Last accessed: Jul 2015.
30. Jim Li, John Chinneck, Murray Woodside, Marin Litoiu, and Gabriel Iszlai. Performance model driven QoS guarantees and optimization in clouds. In *Proc. of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*, pages 15–22, 2009.
31. Awad M. and Daniel A. Menascé. Dynamic Derivation of Analytical Performance Models in Autonomic Computing Environments. In *Proceedings of the 2014 Computer Measurement Group Performance and Capacity Conference (CMG)*, Nov 2014.
32. D. Menascé, Honglei Ruan, and Hassan Gomaa. Qos management in service-oriented architectures. *Performance Evaluation*, 64(7-8):646–663, August 2007.
33. V.S. Borkar M.S. Branicky and S.K. Mitter. A unified framework for hybrid control. In *Proc. IEEE Conf. Decision and Control*, pages 4228–4234, Lake Buena Vista, FL, Dec 1994.
34. Fiona Fui-Hoon Nah. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour and Information Technology*, 23(3):153–163, 2004.
35. Qais Noorshams, Dominik Bruhn, Samuel Kounev, and Ralf Reussner. Predictive Performance Modeling of Virtualized Storage Systems using Optimized Statistical Regression Techniques. In *Proc. of the ACM/SPEC Intl. Conf. on Performance Engineering*, pages 283–294, 2013.
36. Ramon Nou, Samuel Kounev, Ferran Julia, and Jordi Torres. Autonomic QoS control in enterprise Grid environments using online simulation. *Journal of Systems and Software*, 82(3):486–502, March 2009.
37. OMG. Meta Object Facility (MOF) Version 2.5, 2015.
38. G. Pacifici, M. Spreitzer, A. Tantawi, and A. Youssef. Performance Management of Cluster-Based Web Services. *IEEE Journal on Selected Areas in Communications*, 23(12):2333–2343, December 2005.
39. John R Quinlan et al. Learning with continuous classes. In *5th Australian joint conference on artificial intelligence*, volume 92, pages 343–348. Singapore, 1992.
40. Abhishek B Sharma, Ranjita Bhagwan, Monojit Choudhury, Leana Golubchik, Ramesh Govindan, and Geoffrey M Voelker. Automatic request categorization in internet services. *SIGMETRICS Perform. Eval. Rev.*, 36(2):1625, Aug 2008.
41. Simon Spinner, Giuliano Casale, Fabian Brosig, and Samuel Kounev. Evaluating Approaches to Resource Demand Estimation. *Performance Evaluation*, 92:51 – 71, October 2015.
42. Simon Spinner, Samuel Kounev, and Philipp Meier. Stochastic Modeling and Analysis using QPME: Queueing Petri Net Modeling Environment v2.0. In *Proc. of the 33rd Intl. Conf. on Application and Theory of Petri Nets and Concurrency*, pages 388–397, 2012.
43. Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. Analytic modeling of multitier internet applications. *ACM Trans. Web*, 1(1), May 2007.
44. André van Hoorn. *Online Capacity Management for Increased Resource Efficiency of Component-Based Software Systems*. PhD thesis, University of Kiel, Germany, 2014.
45. P. van Overschee and B. de Moor. *Subspace Identification for Linear Systems—Theory, Implementation, Applications*. Kluwer Academic Publishers, Boston-London-Dordrect, 1996.
46. Qi Zhang, Ludmila Cherkasova, and Evgenia Smirni. A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications. In *Proceedings of the Fourth International Conference on Autonomic Computing*, page 27ff, 2007.

Pre-print version for personal use only!