Control Theory for Model-based Performance-driven Software Adaptation

Davide Arcelli
Vittorio Cortellessa
Università dell'Aquila, Italy
{davide.arcelli |
vittorio.cortellessa}@univaq.it

Antonio Filieri
Universitat Stuttgart, Germany
antonio.filieri@informatik.uni-stuttgart.de

Alberto Leva
Politecnico di Milano, Italy
leva@elet.polimi.it

ABSTRACT

Self-adaptive techniques have been introduced in the last few years to tackle the growing complexity of software systems, where a major complexity factor leans on their dynamic nature subject to sudden and unpredictable changes that can heavily impact on the software architecture quality. Non-functional models, as generated from architectural descriptions of software, have been proven as effective instruments to support designers meeting non-functional requirements since the early architectural phases. However, such models still lack of intrinsic support for adaptable software. Goal of this paper is to extend the modeling capabilities to the case of software adaptation aimed at satisfying performance requirements. In particular, we illustrate how control theory can solve the problem of keeping within pre-defined ranges the indices of a Queueing Network model (such as queue length) through software adaptation actions (such as replacing software services with less resource-demanding ones), while the model is subject to disturbances (such as workload and/or operational profile variations). For this goal we first introduce a library of Modelica components that represent Queueing Network (QN) elements with adaptable parameters (that can be used as knobs for adaptation actions). Then we use such components to build "adaptable" QN models that are subject to disturbances. Finally, in the same framework, we introduce controllers that drive the QN adaptation. We demonstrate the soundness of our approach on a simple representative example in two ways: (i) on one end, we provide a formal proof of controller performance guarantees, and (ii) on the other end we show the sensitivity over time of software adaptation actions to different (types and intensities of) disturbances.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques, Performance attributes

QoSA'15, May 4–8, 2015, Montréal, QC, Canada.
Copyright © 2015 ACM 978-1-4503-3470-9/15/05 ...$15.00.
http://dx.doi.org/10.1145/2737182.2737187.

Keywords

Software Performance; Performance Modeling; Control Theory; Software Adaptation

1. INTRODUCTION

The dynamic nature of modern software systems has induced in the last few years new challenges to software engineers, where a relevant one is to enable such systems facing different types of unpredictable changes (e.g., in the software context, in the user requirements).

The introduction of self-adaptive techniques has been proposed to enable software dealing with changes. Adaptation actions and policies are triggered (in a proactive or reactive way) to allow a software system to offer acceptable levels of Quality of Service (QoS), while preserving semantic correctness with respect to its functional requirements. For example, a system might be required to continuously guarantee a prescribed average response time in spite of unforeseen environmental fluctuations; as a violation of this requirement is observed or predicted, the adaption mechanism counteracts the violation, e.g., by providing additional resources [1].

Control theory tackles since decades the challenge of adapting (physical) plants in a variety of engineering domains, and in the last few years it has started to be applied to build adaptable software [2–5]. In principle, adaptable software can be in fact considered a controllable *plant* fitting in the basic feedback control loop scheme [6, 7] of Figure 1.

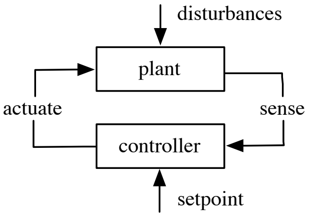

Figure 1: Feedback control loop.

One major advantage of control theory comes from the analytical guarantees it can provide on the controlled system behavior, due to the mathematical grounding of its techniques. However, the intrinsic non-linearity and complexity of software system behaviors is hard to abstract into a suitable mathematical representations, i.e. as system of differential or difference equations, limiting the applicability of control theory in practice. To tackle this difficulty, we follow a paradigm introduced in [5], where the dynamic system of equations describing a software behavior is constructed exploiting an intermediate analytical view of a software architecture as a pivot. Indeed, analytical models such as Markov chains, Petri nets, or Queueing Networks (QNs) are already established in the software engineering practice to abstract QoS-related features since early stages of architectural design. Such models are simpler than design ones, though still informative enough to reason about system QoS. Keeping these models updated at runtime turns them into a current representation of the situation, encapsulating monitoring information into a more useful semantic view [8, 9].

While [5] was concerned with reliability requirements, exploiting discrete time Markov chains as pivot analytical model, in this paper we focus on performance requirements. In particular, referring to Figure 1, we envisage: a QN performance model as a plant, whose sensed values can be any measured performance indexes, whose values are required to satisfy a quantitative constraint (e.g., the end-to-end average response time has to be less than 10ms); *control variables*, whose values can be decided by the controller and enforced on the running system through actuators, are represented by modifiable model parameters (e.g., the routing strategy of incoming requests); the effect of *disturbances*, i.e. external phenomena outside from the system control but affecting its behavior (e.g., workload spikes or fluctuations), are captured as well by changing model parameters, whose values are continuously updated by a monitoring infrastructure [8]. In this paper, we focus on control goals formalized as *setpoint tracking*, i.e. where measured performance indexes have to be kept as close as possible to a reference value provided by the user (the *setpoint*).

The idea of applying control theory in the context of sw/hw systems for sake of performance guarantee is not new [10, 11], and in the last few years some contributions have appeared that combine control theory and design of adaptable software [12, 13]. However, the application of control theory at the software performance modeling side has still many open issues, as outlined in [14], claiming the need to explore adaptive and robust techniques to deal with parameter uncertainty when monitoring does not suffice.

The two main limitations that we aim at addressing in this context are: (i) the lack of an unifying model-based approach to the control of adaptive software driven by performance requirements; (ii) the fact that current approaches to adaptive software often consider as modifiable only hardware variables (e.g., the number of CPU cores), whereas uniformly dealing with hardware and software variables would be important to widen the spectrum of adaptation actions.

Based on the Modelica language [15], we have worked on a unifying framework aimed at overcoming these limitations. In particular, the contributions of this paper are: (i) a library of Modelica components that represent QN basic elements (e.g., queue, server, workload); (ii) the design of controllers in Modelica for QN models subject to disturbances of their software and hardware parameters; (iii) a proof of performance guarantees provided by a controlled QN model; (iv) an empirical evalution showing the behavior of a model and a controller under different scenarios.

In perspective, our work aims at defining a broadly applicable methodology for the design of control systems based on QN models with formally provable quality guarantees. This would both support the development of the system and controller since early stages of design and drive the implementation of controllers to manage the runtime behavior of the deployed system.

2. CONTROL THEORY FOR SOFTWARE PERFORMANCE

Control systems are backed by a *control theory* that undergoes the mathematical tools supporting the definition of controllers with *formally proved* quality properties [16, 17]. The design of a controller is based on a mathematical model of the controlled system behavior. This model is usually a *dynamic model* defined by differential or difference equations. The dynamic model formalizes the relationships between the time, the *system state*, the *control variables* (i.e., the system inputs we can enforce), and the *controlled variable* (i.e., the system output we want to effect).

Besides the effects of the control variables, the system model is required to take into account possible *disturbances*, i.e., the effect of external phenomena affecting its behavior, which we can at most measure but not directly affect. A broad variety of disturbance types have been studied to characterize physical phenomena as well as to represent errors or uncertainties about the system and the monitors [17].For software performance, the workload and operational profile an application is subject to depend on the user multiplicity and behavior; it is possible to formally characterize them but, in general, they fall out of the application control. For this reason, we will here consider them as disturbances.

A proper dynamic model of the controlled system and the relevant environmental phenomena allows the application of a broad variety of (more or less automatic) techniques for the design of controllers enjoyng several important qualities [16, 17]. For this work, we will focus on the synthesis of *setpoint-tracking* controllers, i.e., controllers designed to keep the controlled variable (e.g., the average response time) as close as possible to a target value (e.g., 300ms).

For sake of soundness, the controllers we will design have to guarantee the following properties:

- *Stability*: A control system is asymptotically stable if, under reasonable assumptions on the initial state, the system will tend to an equilibrium point; i.e., for any given input, the output converges to a specific value (within a convenient accuracy). In the case of setpoint-tracking, whenever the set point is reachable, it has to be the equilibrium the system tends to.
- *Low settling time*: The time to converge to the setpoint (or the closest feasible equilibrium) has to be kept conveniently short.
- *Robustness*: The control system is required to converge to the setpoint despite both the effect of disturbances and possible inaccuracies in the dynamic model. To achieve this goal we will focus here on *feedback control*, which allows the controller to overcome both disturbances and inaccuracies by learning from the effects of its past own actions to refine its strategy [7].

Controlling Software Systems. Despite the undisputed benefit of formally guaranteed control, the application of control theory to software is fairly limited [2]. Indeed, unlike physical phenomena, software behavior is hard to model by means of dynamic systems of equations. The algorithmic nature of a program often leads to the introduction of complex non-linearities in the models, thus reducing their suitability for control design. On the other hand, software architecture to a larger extent can often be conveniently abstracted by analytical models (such as Queueing Networks and Markov Models [18]) able to capture the main quantitative measures of interest.

Although most of the analytical models used to describe software performance are not straightforwardly mappable to a dynamic system of equations, they may be exploited to fill the semantic gap between the software architecture models/artifacts and the equations, as firstly proposed in [5]. The use of established analytical models as "pivot" for the generation of dynamic systems of equations has a twofold benefit: (i) it simplifies the construction of the dynamic model by providing a more succinct and precise quantitative view on the system; (ii) it broadens the applicability of the controller design methodology to every system formalizable through the intermediate model. For example, defining a general control design methodology for QNs would allow the construction of controllers for a variety of systems whose performance concerns can be captured through a QN model.

3. CONTROL APPROACH

The first step to define a control methodology for QN models is the provisioning of a suitable mean for their formalization as dynamic systems. To this end, we implemented a first version of a Modelica library for QNs. Modelica [15] is a modeling and simulation environment widely used by control practitioners to define and study dynamic models of physical phenomena and engineered systems, and to support the design and synthesis of suitable controllers. Our library includes at today only basic QN component types, that are: queues, service centers, routing nodes, and workload generators (both deterministic and probabilistic). Instances of these types can be created and connected together to seamlessly draw a QN model. Each of the component types contains peculiar inputs/outputs and state equations. The former determine the interaction with the other components, while the latter represent a (parametric) dynamic model specified as a system of differential equations that capture the time behavior of the component instance, according to its initial state and the input it receives.

When a QN component is instantiated, the designer just needs to set the desired parameters (e.g., the service rate) and the connection to other components (e.g., reflecting the architectural control flow). Furthermore, Modelica is an object-oriented framework, with constructs for type hierarchies and inheritance. Hence our library is easily extendable by either adding other first class types defined in the QN literature (e.g., fork/join nodes, passive resources) or by extending existing types with additional features (e.g., service centers with special scheduling policies).

In this section we describe the main currently supported features through the formalization, control, and analysis of an example application: the software and the performance model are described in Section 3.1.1 and 3.1.2, respectively; Section 4 describes the component types used to formalize an application with our Modelica library. In Section 5 we illustrate the control strategy and we demonstrate its formal properties that are empirically analyzed in Section 6.

3.1 Running Example

3.1.1 Software model

The application scenario of our control approach is based on a software/hardware system consisting of a web application that provides itineraries of interest to the users. We introduce two basic concepts in this context: (i) *Locations*, that represent geographical places, e.g., cities or quarters of a metropolis; (ii) *Points of Interest* (PoIs), that represent specific places of locations that users would be interested to visit, e.g., museums, parks, hotels.

A typical feature of the considered web application is the calculation of an itinerary from a location s to a location d, possibly including PoIs of certain types that can be specified by the user. Figure 2 shows an UML Sequence Diagram describing the dynamics of the use case scenario we consider that involves the above feature.

Actors and component instances involved in the scenario are the following:

- *User*: it represents any user of the application (e.g., a tourist). It uses the application through a *Browser* which renders a web interface that can provide four different levels of user experience, which we report in a descending order with respect to the quality perceived by the user: *High*, *Medium-High*, *Medium-Low*, *Low*.
- *DBMS*: it represents the Data Base Management System used by the application.
- *Web server*: it is implemented as a 3-layer architecture, hence it consists of: (i) a presentation layer (wpl) that provides the web interface downloaded and rendered by the *Browser*; (ii) a business layer (wbl) for computing itineraries; (iii) a data layer (wdl) that interacts with the *DBMS*.

The dynamics that take place in the considered scenario are described in the following. Given an user's itinerary request, an itinerary from a location s to a location d is first calculated without considering any PoI type $t \in T$ possibly specified by the user (messages from 1 to 4). At this point, with a certain probability, the user's original request expected that all the PoIs whose type belongs to T (e.g., museums) would be added to the itinerary. Hence, for each PoI type $t \in T$, wbl requests the PoIs of type t, that are retrieved by the *DBMS* and then returned to the wbl (messages from 5 to 9). Subsequently, the wbl calculates the itinerary it from s to d, also including some intermediate locations among the ones with PoIs in P (message 10). Without loss of generality, we can assume that location inclusion strategies are codified in the itinerary calculation algorithm. The average number of PoI types specified by a user (i.e., $|T|$) determines the average number of iterations N of the loop. It can be demonstrated that $N = p/(1-p)$, where p is the average probability of doing a further iteration, hence $p = N/(N+1)$. After N iterations, based on the level of user experience that has to be provided, the calculated itinerary is returned to the *Browser* that renders the web interface (messages from 11 to 13).

The amount of computational resources (i.e., the service demand) required by the web server layers varies, based on the provided user experience quality. In particular, we assume that: if the latter is *High*, then the system provides a map where the itinerary is shown and navigable; in case of *Medium-High* experience, it provides a map where the itinerary is shown but not navigable; in case of *Medium-Low* one, it provides a graph representation of the itinerary; finally, in case of *Low* experience, it just provides the itinerary as a list of locations. Hence, higher levels require larger computational costs to the web server. One might be interested to differentiate service levels even on the *DBMS*, for example in terms of amount of information concerning a PoI, as follows: higher experience needs more information than lower levels. Nevertheless, for sake of simplicity we do not consider service level differentiations for the *DBMS*.

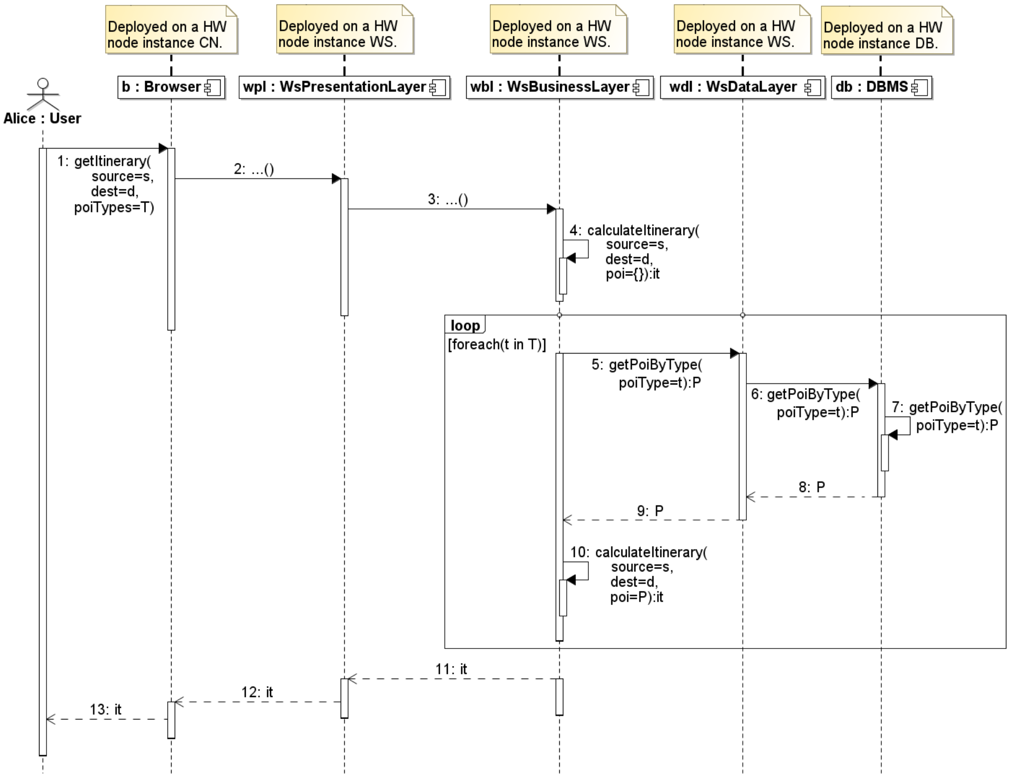
Deployed on a HW node instance CN.
Deployed on a HW node instance WS.
Deployed on a HW node instance WS.
Deployed on a HW node instance WS.
Deployed on a HW node instance DB.
Alice : User
b : Browser
wpl : WsPresentationLayer
wbl : WsBusinessLayer
wdl : WsDataLayer
db : DBMS
1: getItinerary(source=s, dest=d, poiTypes=T)
2: ...()
3: ...()
4: calculateItinerary(source=s, dest=d, poi={}):it
loop
[foreach(t in T)]
5: getPoiByType(poiType=t):P
6: getPoiByType(poiType=t):P
7: getPoiByType(poiType=t):P
8: P
9: P
10: calculateItinerary(source=s, dest=d, poi=P):it
11: it
12: it
13: it

Figure 2: Software and hardware characterization of the running example.

Note that this software characterization of our running example represents a broad spectrum of adaptable software architectures, where services can be offered with different quality levels in order to satisfy service level agreements without violating the functional requirements.

Figure 2 also describes the hardware characterization of our example, i.e., hardware nodes and deployment of software services (see lifeline annotations). In particular, there are three hardware node instances: (i) *CN*, where the user's *Browser* is deployed; (ii) *WS*, where all the artifacts manifesting the web server services are deployed; (iii) *DB*, where the *DBMS* is deployed.

Given the described deployment, it is important to note that all the artifacts deployed on the web server (i.e. *wpl*, *wbl*, *wdl*) share the same, limited, computational hardware resources. As previously said, the *High* level of user experience requires a higher computational effort to the web server than the other levels. Hence, *High* has to be preferred whenever the available *WS* computational resources allow to run it without violating performance requirements. However, it may happen that there are not enough resources to enable that level for all the users. In such case some users are tentatively served with a *Medium-High* level that induces a higher service rate due to a lower computational effort. The same applies to the remaining service levels, i.e. *Medium-Low* and *Low*. Essentially, the web server implements a "best-effort" strategy aimed at providing the best user experience with respect to the available *WS* computational resources, while satisfying the performance requirements.

3.1.2 Performance model

Following established literature on architecture to performance model transformations, an open QN model is generated for this example, reported in Figure 3 in Modelica notation (where blue arrows represent inputs and white ones represent outputs). Besides typical QN elements (such as service centers and their queues, i.e. WS and DB), a dark block has been added to represent the WS controller (i.e. C_{WS}), as well as other simple blocks appear to represent parameters, setpoints, and disturbances[1]. We call *Adaptive QN* (AQN) this extended model, described in the following.

The workload generator is modeled by the disturbance block labeled as r_{in} that represents the rate of arrivals varying over time in an unpredictable way. A job, after being properly processed by the QN, exits at the output arrow labeled as r_{out} that represents the system throughput.

A second disturbance that we consider is the *branching* probability p_o, i.e., the average probability that a request exits the QN after being served by the *WS*. It relates to the probability of doing further iterations p defined in Figure 2 as follows: $p_o = 1 - p$. Hence, p_o is the average probability that there are no other types of PoI to consider in the itinerary computation after a request has been served by the *WS*. Therefore, p_o as a disturbance corresponds to variations of this significant aspect of the operational profile. Similarly to the workload, p_o varies over time in an unpredictable way.

The application of our control approach to the running example results in the synthesis of the control block of Figure 3, i.e., C_{WS}, with its corresponding inputs and outputs. Controllers are here part of the system itself, thus in fact enabling service centers to behave as required.

As previously said, we assume in the back-end that the amount of computational resources requested to *WS* varies, based on the experience provided to the user. The parameter

[1] Since we are interested in the web application back-end, we do not introduce a service center representing *CN* that belongs to the front-end.

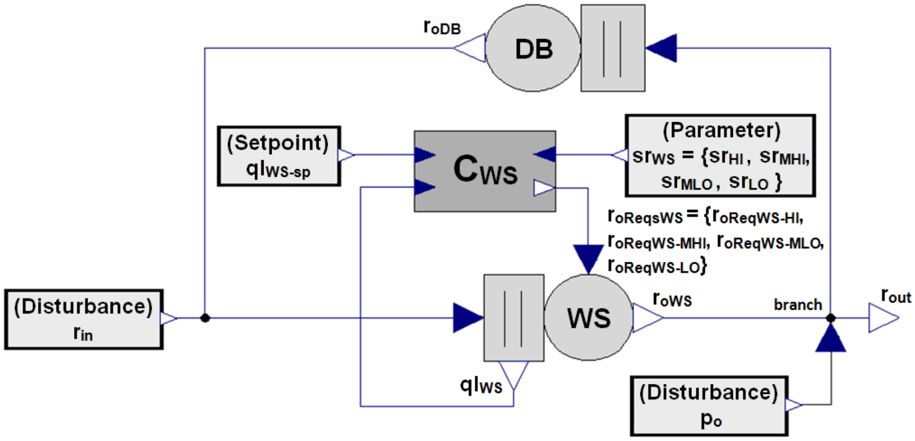
roDB
DB
(Setpoint)
qlWS-sp
CWS
(Parameter)
srWS = {srHI, srMHI, srMLO, srLO}
roReqsWS = {roReqWS-HI, roReqWS-MHI, roReqWS-MLO, roReqWS-LO}
(Disturbance)
rin
WS
roWS
branch
rout
qlWS
(Disturbance)
po

Figure 3: Adaptive QN of the running example.

block of Figure 3 defines a set of inputs to the controller that represent *WS* processing capabilities sr_{WS}, namely the maximum *service rates* (in jobs per second) for the four different levels of user experience that can be provided by *WS*. For example, $sr_{WS} = \{5000, 10000, 20000, 30000\}$ means that in one second *WS* is able to process up to: 5000 *High* service level jobs, or 10000 *Medium-High* ones, or 20000 *Medium-Low* ones, or 30000 *Low* ones.

Goal of the controller is to maintain a sensed performance index, i.e., the *WS queue length* ql_{WS}, equal to its target value ql_{WS-sp} defined as input from the setpoint block. In order to achieve this goal, the controller determines the share of jobs to be served at each service level, based on the sensed value of ql_{WS}, as a vector $r_{oReqsWS}$ of shares (i.e., $\{r_{oReqWS-HI}, r_{oReqWS-MHI}, r_{oReqWS-MLO}, r_{oReqWS-LO}\}$). Hence, C_{WS} continuously adjust $r_{oReqsWS}$ at runtime, thus giving a portion of computational resources to the service levels in order to maintain the targeted queue length. This goal has to be achieved notwithstanding the continuous variation of the disturbances r_{in} and p_o.

For example, let us assume that, at time t, *WS* is serving with *High* service level all the incoming jobs. This means that C_{WS} is requiring to *WS* an output rate $r_{oWS} = r_{oReqWS-HI}$ for 100% of jobs, which are thus being served with sr_{HI}. Now, suppose that the workload rapidly increases from t to $t + \delta t$. In this interval, when the *High* service level cannot be maintained to serve all the incoming jobs without violating the queue length setpoint, a number of jobs is served with a lower level of user experience, i.e. sr_{MHI}. This mechanism further reiterates towards lower levels, i.e. sr_{MLO} and sr_{LO}, if needed. The opposite mechanism is applied in case of workload decreases. In fact, when lower active service levels are not needed because the queue length has been relieved, then a number of jobs is served with higher service levels.

As outlined before, this adaptation schema copes with many common scenarios where the same functional service can be provided with different levels of quality. Note that, with our approach, the quality of service does not necessarily change for all users, but we are able to model the contingency where (at the same time) different requests are processed at different levels of service. In the performance modeling domain this corresponds to a specific scheduling policy of a multi-class service center (see Section 4).

Summarizing, there are some important aspects that make significant the example we have considered, with respect to the problem of performance model adaptation through setpoint-tracking controllers:

- We consider software and hardware aspects within the same modeling framework, hence we can capture the relationships among decisions at hardware and/or software level. Indeed the available hardware resources represent a constraint to be considered by software-level decisions. On the other hand, one may decide to stress certain features more than others at the software level, thus increasing or decreasing the demand of hardware resources of services. In our example, this pattern occurs for *wpl* that has to provide the best level of user experience at a certain rate that guarantees the performance requirement satisfaction, under limited resources of *WS*.
- Current decisions in a service center may affect the future decisions on the same center and/or on those it interacts with. This problem is quite common even in simple topologies and may lead to destabilizing chain effects. In our example, this is captured by the loop from the *WS* to the *DB* and back to the *WS*, as follows: if the *WS* sends too many requests to the *DB*, then the latter's output will contribute to increase the requests to the former in the future.
- Several external parameters (i.e., disturbances) may change at runtime. In particular: (i) the workload intensity, i.e., the volume of incoming requests; (ii) the operational profile, i.e., the probability that a request leaves the system after being served by the *WS*.

4. A MODELICA LIBRARY FOR ADAPTIVE QUEUEING NETWORKS

In this section, we describe how the components of the AQN models of Figure 3 have been formalized through Modelica library elements.

- *Queue.* The state of a queue is defined as the number of enqueued requests. An initial occupation level can be set as a parameter of the queue instance. It is possible to push as many requests as desired and to pop an arbitrary number of requests, less or equal than those enqueued.
- *Processing unit.* A processing unit is a stateless component which can pop elements out of a connected queue and process them. Its performance is defined by a real valued positive processing rate.

- *GPS service center.* A service center is composed by one queue and one processing unit. The only scheduling policy that we have devised up today is a Generalized Processor Sharing (GPS) one [19]. In particular we envisage that a (non-uniformly distributed) share of processing is held by each class of jobs, where different classes of jobs may have different resource demands. This is a general modeling approach to represent the case of a service center that processes jobs by the same "functional" type (e.g., jobs that represent the same software function/operation) that can be partitioned in classes, where each class provides a different quality level. From a software viewpoint, this is a scenario (as illustrated in our running example) where different adaptations are available for a certain operation, and each adaptation requires a different amount of resources. The processing shares among classes of jobs are regulated by the controller.

  Both *WS* and *DBMS* of our example application are modeled as GPS service centers. In particular, the former exhibits four different service levels corresponding to the alternative levels of user experience; the latter is a special case instance having only one class of jobs.
- *Branch node.* It represents a stochastic branching of jobs, where a distribution function regulates the probability that a job is routed along a certain outgoing path. In the example, it corresponds to the branching point where a job can either leave the system or be routed to the database, after being processed by *WS*.
- *Workload.* A workload is represented by either a function associating to each time point a number of generated requests or a probabilistic distribution (possibly varying over time) from which the number of incoming requests is sampled. Both features are implemented on top of Modelica basic components describing real and integer functions or probability distributions. In the example, we used a function block to simulate different change patterns of the incoming workload.

5. CONTROLLER DESIGN

In this section, we sketch the main steps leading to the synthesis of a controller, taking as example the system introduced in Section 3.1.

Control requirements. The system is required to process all the incoming requests, using as much as possible the highest quality service level and withdrawing toward lower levels only if needed. The processing resources allocated for the web server are limited; the controller can only distribute them among the available ones, without allocating more. If additional resources are provided at runtime, the controller has to adjust its policies to exploit all of them. The control has to be robust to changes in the operational profile (e.g., the probability that a request leaves the system after being processed by the web server). The controlled system has to stably achieve its goals whenever feasible, or go as close as possible to their satisfaction (e.g., steadily keep the maximum processing rate to provide the best performance, though insufficient to achieve the goal).

Modeling a queue. Denoting by $r_i(t)$ and $r_o(t)$ the input and output rate of a node and by $n(t)$ the occupation of its queue, t indicating the continuous time, the dynamic system describing its behavior is:

$$\dot{n}(t) = r_i(t) - r_o(t), \tag{1}$$

where the dot indicates the time derivative. A network with N nodes is naturally described as a continuous-time dynamic system of order N, whose state variables are the queue occupations $n_k(t)$, $k = 1 \dots N$. Such a system takes the form:

$$\dot{\mathbf{n}}(t) = \mathbf{B_o}(\mathbf{p}(t))\mathbf{r_o}(t) + \mathbf{B_i}\mathbf{r_i}(\mathbf{t}) \tag{2}$$

where the bold face distinguishes vectors and matrices. Notice that matrix $\mathbf{B_o}$ depends on some vector $\mathbf{p}(t)$ of time-varying routing probabilities for the nodes' output rates, which makes the system Linear, Parameter-Varying (LPV) [17]; matrix $\mathbf{B_i}$, accounting for the effect of external input rates, is conversely constant. For example, the system in Figure 3, when put in the form (2), reads:

$$\begin{bmatrix} \dot{n}_{WS}(t) \\ \dot{n}_{DB}(t) \end{bmatrix} = \begin{bmatrix} -1 & 1 \\ 1 - p_o(t) & -1 \end{bmatrix} \begin{bmatrix} r_{oWS}(t) \\ r_{oDB}(t) \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} r_{in}(t). \tag{3}$$

Any LPV system can be represented in the standard form (see [20] for a comprehensive overview):

$$\dot{\mathbf{x}}(t) = \mathbf{A}(\theta(t))\mathbf{x}(t) + \mathbf{B}(\theta(t))\mathbf{u}(t), \tag{4}$$

where $\mathbf{u}(t)$ represents the control inputs (e.g., the number of requests to be served at each service level by WS), $\mathbf{x}(t)$ the state (e.g., the queue occupancy), and $\theta(t)$ a set of varying parameter (e.g., the resources required to WS to serve a request for each service level, or the exit probability p_0).

With respect to this standard form for LPV, the system (2) has some relevant peculiarities:

- the dynamic matrix $\mathbf{A}$ is identically zero, as the state $\mathbf{n}$ does not appear on the right hand side of (2);
- there are two $\mathbf{B}$ matrices, one for the internal output rates – that will soon take the role of the control actions – and one for the exogenous disturbances provided by the external input rates (this is conceptually irrelevant, since one could join $\mathbf{B_o}$ and $\mathbf{B_i}$ in a single non-square matrix $\mathbf{B}$ as per (4), but useful in practice);
- the only varying matrix is $\mathbf{B_o}$;
- however the diagonal elements of $\mathbf{B_o}$ are all constant and equal to -1,
- while its off-diagonal ones are either zero, or one, or – if varying – probabilities, thus in the range $[0, 1]$.

A first important remark concerns the achievable stability properties. The key point here is that the varying vector – $\mathbf{p}(t)$ in our case – lies in a convex hull. To prove this it suffices to observe that any binary routing gives rise to a couple of elements in the form $(p_j, 1 - p_j)$, thus to a constraint surface given by a plane orthogonal to the axis of the only free probability p_j in the vector space of $\mathbf{p}$. Any m-ary routing, conversely, corresponds to $m-1$ independent probabilities bound by their sum not exceeding the unity, thus to a hyperplane intersecting their axes, each in its point $+1$.

As such, see again [20] for the proof, any feedback control system containing (2) and either a linear constant-parameter controller, or a LPV one that however does not introduce additional varying entities in addition to $\mathbf{p}$, is guaranteed asymptotically stable if those corresponding to the vertices of the hull are asymptotically stable, and with the same degree of stability. In other words, denoting by $\{\mathbf{A}_{cl,v}\}$,

$v = 1 \dots V$ the set of V dynamic matrices for the closed-loop systems in the hull vertices, to guarantee LPV stability it is required to find a single matrix $\mathbf{P} > 0$ such that

$$\mathbf{P}\mathbf{A}_{cl,v} + \mathbf{A}_{cl,v}^T\mathbf{P} < 0, \qquad v = 1 \dots V. \tag{5}$$

A method to solve the problem is reported in [21], to which the interested reader is referred, as further details would stray from the scope of this paper. For our purposes, however, a few points are worth noticing. First, since the $\{\mathbf{A}_{cl,v}\}$ matrices depend on the controller, (5) provides a conceptual means for its synthesis. Then, the existence of a solution for (5) is not guaranteed in general. However, according to results on switching systems – a category closely related to LPV ones – a sufficient condition can be stated as requiring the eigenvalues of the closed-loop dynamic matrices to be real negative for any value of $\mathbf{p}$ [22]. Further research is required on this, especially to avoid unnecessary conservatism, but at present no network examples were found where the condition cannot be fulfilled. Finally, although finding a solution for (5) does provide a controller, this is not the most practical way to go, particularly as far as the physical interpretability of the involved design parameters is of concern. However, we present some practical alternatives in the following. We refer to the presented case study for simplicity, deferring a general treatment of the matter to future works.

A second remark regards the possibility of achieving the desired properties – stability, see above, and performance, dealt with later on – by *only* measuring queue occupations, which can be done precisely and efficiently. Should specifications involve throughputs, for example, this allows to avoid any rate measurement, which may be critical to obtain and invariantly requires some arbitrary choices on how the measurement is taken.

Controller design. We aim at designing a control system having a minimal impact on the existing system:

- we set up a totally decentralized scheme, where each node has its own controller and said controllers do not communicate with one another,
- each controller acts on $r_o(t)$ to keep $n(t)$ for its node at a convenient reference value $n^o(t)$ (which can be as close to 0 as desired),
- no reliable estimation of probabilities is assumed for dispatching nodes, thus all the controllers are fixed-parameter (i.e. the system has to rely solely on robustness with respect to unforeseen parameter variations).

We start by noticing that the transfer function computed from (1) (which describes the behavior of a queue element), denoting by s the Laplace transform complex variable [17], is:

$$N(s) = \frac{1}{s}\big(R_i(s) - R_o(s)\big). \tag{6}$$

Endowing this system with a feedback controller $C(s)$ to regulate n acting on r_o results in the block diagram of Figure 4. Thus, the transfer function from $r_i(t)$ to $r_o(t)$ is:

$$\frac{R_o(s)}{R_i(s)} = -\frac{C(s)N(s)}{1 - C(s)N(s)}. \tag{7}$$

With a purely proportional local controller – i.e., one with transfer function $C(s) = -K$ ($K > 0$, as to increase n one has to decrease r_o) – (7) becomes

$$\frac{R_o(s)}{R_i(s)} = \frac{1}{1 + s/K} \tag{8}$$

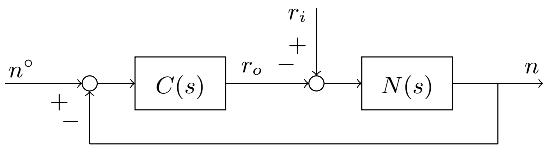

Figure 4: Block diagram for a local control loop.

and the settling time for a step variation of $R_i(t)$ is five times the dominant closed-loop time constant, i.e. $5/K$. The proportional law is used here as it is apparently the simplest one, and allows to easily illustrate stability-related facts. More complex laws may be introduced, like the PI, but their formal analysis would be too long to be reported here. A PI controller will be empirically evaluated in Section 6. Notice that a controller of form (8) is required for each queue to guarantee the corresponding processing units will process all the incoming request by keeping the queue length to its setpoint. For example, as mentioned in Section 3.1.1, the DBMS in our example requires a controller trivially prescribing the processing of all the requests at the single quality level assumed for the DBMS. Such a controller is required by the operational semantics of our Modelica-based AQNs, but it does not alter the typical semantics of the GPS service center.

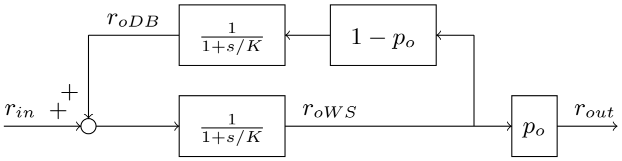

Figure 5: Block diagram for the example system.

Coming back to the system of Figure 3, the overall system can be described by a time-varying block diagram, where however the dynamic blocks are LTI in the form (8). The diagram is shown in Figure 5. Notice that at node level there is neither modeling uncertainty nor measurement error, as the system is exactly described by (6), and the sole measurement required is the occupation of the local queue.

In the block diagram just mentioned, only static blocks are time-varying, and come in tuples with gains p_i summing to the unity. The presented example, adopting the same K for the web server and the database as there is no reason to do anything different, results therefore (see again Figure 5) in a dynamic relationship from the network input rate to its output one given by

$$\frac{R_{out}(s)}{R_{in}(s)} = \frac{\frac{p_o}{1+s/K}}{1 - (1-p_o)\left(\frac{1}{1+s/K}\right)^2} = \frac{1 + s/K}{1 + \frac{2}{Kp_o}s + \frac{s^2}{K^2p_o}}. \tag{9}$$

The poles of (9) – i.e., the roots of its denominator – are $-K\left(\sqrt{1-p_o}+1\right)$ and $K\left(\sqrt{1-p_o}-1\right)$, both real and negative under the assumed hypotheses, independently of p_o. Thus, LPV stability is ensured [17, 20], and performance (quantified by the settling time mentioned above) can be tuned by adjusting K. Notably, a lager value for K leads to faster reactions to fluctuations and a shorter settling time, however, such fast reaction might lead to unnecessary overshooting in case of transitory variations of the workload or monitoring outliers. The use of a PI controller can in this case improve the robustness of the controller. The use of a PI will be evaluated empirically in Section 6.

As a final remark about the stability of the controlled system is that one of the poles tends to zero (from the left) as $p_o \to 0$. This leads the system to the stability limit but refers to an apparently pathological case, as if no job exits the network, it is clearly impossible to control any queue occupation. Nonetheless, when no requests are being processed, there is no actual need for specific control actions (i.e. the controller will not require any processing; this is achieved with a standard antiwindup bounding [17]).

6. EVALUATION

An experiment is here reported where the AQN of Figure 3 is subjected to time-varying r_{in} and p_o disturbance profiles shown in Figure 6, for a simulated time span of one day. Although the purpose here is to show the viability of the proposed modeling and control approach, care was taken to provide a realistic input rate, with both long- and short-term variability plus a couple of job hauls caused for example by crowding, and to have significant variations in the probability of a job exiting the system.

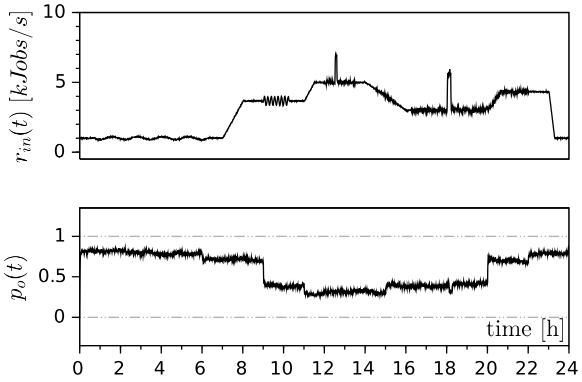

Figure 6: Disturbance profiles.

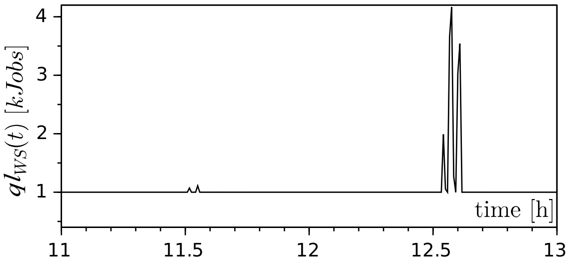

Figure 7: Web server queue length detail under heaviest load (see Figure 6).

The adopted control solution is the decentralized one, with fixed-parameters PI controllers (as there is no space to discuss other possibilities). This means that each node acts independently, requiring only a measurement of its queue's occupation. Both the *WS* and the *DB* queue occupation setpoints were set to 1000, while both PIs have $K = -0.5$ (see Equation (9)). Tuning was achieved by settling time prescription, see [23] for a wealth of suitable techniques.

The *WS* queue setpoint is kept almost perfectly throughout the test, while in three cases (the toughest shown in Figure 7) the system cannot sustain the input rate, and the web server queue temporarily runs away. However, as soon as a manageable situation is recovered, the controller regains the set point quickly. Hence, the system tends to the setpoint whenever possible (stability) and it does so in a reasonably short time – 5 minutes in the worst case – (low settling time), overcoming both disturbances and inaccuracies (robustness). As a result, see Figure 8, the output rate satisfactorily matches the incoming requests.

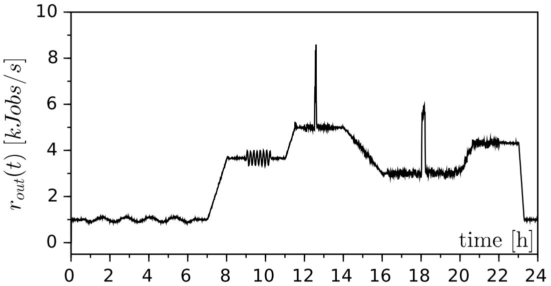

Figure 8: Achieved output rate.

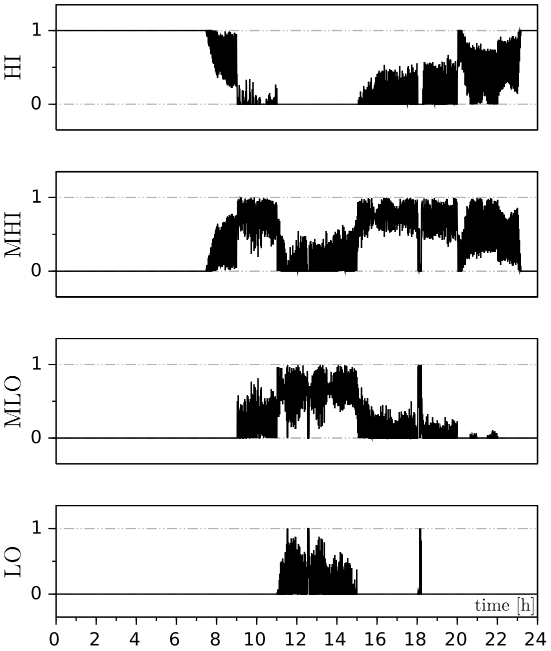

Figure 9: Web server share among the four service levels.

Figure 9 finally shows the share of the *WS* requests handled by the various service levels. Four were introduced (i.e., HI, MHI, MLO and LO) in decreasing quality order that exhibit, in the same order, an increasing maximum throughput at full computational resource share, as previously introduced. It can be noticed that the system correctly employs the best quality service level when possible, and progressively brings in lower quality ones when this is required by the current load.

Summarizing, the test shows that the proposed modeling paradigm is suitable for the assessment of a control strategy at the abstraction level it is intended for.

The evaluation presented in this section is obviously related to our example context that has been limited to a model with one controller. AQNs with multiple controllers represent our next research focus, where the behavior of different controllers is not independent from each other. In such context, coordination among controllers have to be introduced to take into account that such dependencies.

7. RELATED WORK

Adaptation is becoming a key concern in software applications [24]. An adaptive application must select from many configurations the one that is most appropriate to satisfy some specific (performance) requirements. There are many examples, from hardware to software development. The evaluation of a new microprocessor design requires studying the impact of input data sets and workload composition [25]. Compiler-level advancements have been developed to support adaptive implementations for performance [26, 27] or power [28]. Other examples come from High Performance Computing, where it is common to change an application parameter to adapt a running application. In [29], a study on tuning Fast Fourier Transformations on graphic processing units is presented, whereas Rahman et al. [30] studied the effect of compiler parameters on both performance and power consumption for scientific computing.

Self-management techniques are also prominent in industry, e.g., in projects like the K42 Operating System [31] by IBM, the Automatic Workload Repository [32] by Oracle, and the RAS Technologies for Enterprise [33] by Intel.

Control theory [6, 7] is capturing an increasing interest from the software engineering community that looks at self-management as a means to meet QoS requirements despite unpredictable changes in the execution environment [2]. Examples of this trend can be found in control of web servers [34, 35], data centers and clusters management [36, 37], operating systems [38–40], and across the system stack [41].

The application of control theory in software engineering, however, is still in a very preliminary stage. Developing accurate system models for software is in fact hard. Moreover, strong mathematical skills are needed in order to deal with complex non-linear dynamics of real systems [42, 43]. These difficulties usually lead to the design of controllers focused on particular operating regions or conditions and ad-hoc solutions that address a specific computing problem using control theory, but do not generalize [44–46]. For example, in [47] the specific problem of building a controller for a .NET thread pool is addressed. More in general, the approach presented herein has the peculiarity of not just closing control loops *around* an existing system, i.e. adding a control layer *on top* of a fully functional one. Controllers are here part of the system itself, thus enabling elements to provide the required functionality. The interested reader can find in [48, Ch. 1] a discussion on the benefits and the new design challenges that such an approach brings into the *arena*.

All the work discussed up to this point aims at controlling running adaptable applications. In this paper, we raise the level of abstraction introducing a *model-based* performance control of adaptive software based on analytical abstractions of a system architecture. In this domain, some effort has been spent to raise adaptation techniques driven by performance (or more general QoS) requirements at the software architecture level [49], where adaptive verification techniques have been also studied [1, 50]. Adaptation approaches for specific architectural paradigms have been introduced, such as Service-Oriented-Architecture [51], as well as for specific Architectural Description Languages, such as Palladio Component Model [52]. An interesting work has been recently introduced in [53] for automatically extract adaptive performance models from running applications. However, none of these papers applies control theory for the control of performance models.

8. CONCLUSIONS

In this paper, we defined a model-based software adaptation approach based on performance models extracted from software architectures grounded on control theory, suitable for ensuring the continuous satisfaction of performance requirements. We developed a preliminary version of a Modelica library of components for representing QN elements with adaptable parameters. This library can be used to formalize QN models subject to disturbances. We described the design of setpoint-tracking controllers for adaptive QNs within the same framework, and we proved the stability of these controllers by reducing them to standard form of LPV controllers (whose properties are well studied in control theory). We empirically showed the robustness of the controlled system to different types and intensities of disturbances.

Starting from the preliminary results reported in this paper, our work aims at defining a broadly applicable methodology for the design of control systems based on QN models with formally provable quality guarantees. In this regard, we plan to consider (among other) multiple performance requirements at the same time, which can possibly conflict one another. Finally, we plan to explore the application of optimal control for implementing policies aiming at achieving the required performance goals while optimizing related system qualities, e.g., minimizing energy consumption.

9. REFERENCES

[1] R. Calinescu et al. "Adaptive model learning for continual verification of non-functional properties". In: *ICPE*. 2014, pp. 87–98.

[2] T. Patikirikorala et al. "A systematic survey on the design of self-adaptive software systems using control engineering approaches". In: *SEAMS*. 2012, pp. 33–42.

[3] A. Filieri et al. "Software Engineering Meets Control Theory". In: *SEAMS*. IEEE, 2015.

[4] J. Hellerstein et al. *Feedback Control of Computing Systems*. 2004.

[5] A. Filieri et al. "Self-adaptive software meets control theory: A preliminary approach supporting reliability requirements". In: *ASE*. 2011, pp. 283–292.

[6] J. L. Hellerstein. "Self-Managing Systems: A Control Theory Foundation". In: *ECBS* (2005), pp. 708–708.

[7] J. Doyle et al. *Feedback control theory*. 1992.

[8] A. Filieri et al. "A formal approach to adaptive software: continuous assurance of non-functional requirements". In: *Formal Aspects of Computing* 24.2 (2012), pp. 163–186.

[9] A. Filieri et al. "Lightweight Adaptive Filtering for Efficient Learning and Updating of Probabilistic Models". In: *ICSE*. IEEE, 2015.

[10] T. F. Abdelzaher et al. "Performance Guarantees for Web Server End-Systems: A Control-Theoretical Approach". In: *IEEE Trans. Parallel Distrib. Syst.* 13.1 (2002), pp. 80–96.

[11] S. S. Parekh et al. "Using Control Theory to Achieve Service Level Objectives In Performance Management". In: *Real-Time Systems* 23.1-2 (2002), pp. 127–141.

[12] A. Filieri et al. "Automated Design of Self-adaptive Software with Control-theoretical Formal Guarantees". In: *ICSE*. ACM, 2014, pp. 299–310.

[13] A. Gambi et al. “Kriging Controllers for Cloud Applications”. In: *IEEE Internet Computing* 17.4 (2013), pp. 40–47.

[14] T. F. Abdelzaher et al. “Feedback Performance Control in Software Services”. In: *IEEE Control Systems Magazine* 23.3 (2003), pp. 74–90.

[15] P. Fritzson et al. “Modelica – A unified object-oriented language for system modeling and simulation”. In: *ECOOP*. Vol. 1445. LNCS. Springer, 1998, pp. 67–90.

[16] K. J. Astrom et al. *Computer controlled systems: Theory and design*. 1994.

[17] W. Levine. *The control handbook*. 2005.

[18] G. Bolch et al. *Queueing networks and Markov chains - modeling and performance evaluation with computer science applications*. Wiley, 2006.

[19] S. Aalto et al. “Beyond processor sharing”. In: *SIGMETRICS Performance Evaluation Review* 34.4 (2007), pp. 36–43.

[20] J. Shamma. “An overview of LPV systems”. In: *Control of linear parameter varying systems with applications*. Springer, 2012, pp. 3–26.

[21] D. Liberzon et al. “Common Lyapunov functions and gradient algorithms”. In: *IEEE Transactions on Automatic Control* 49.6 (2004), pp. 990–994.

[22] A. Leva et al. “Tuning of event-based industrial controllers with simple stability guarantees”. In: *J. of Process Control* 23.9 (2013), pp. 1251–1260.

[23] K. Åström et al. *Advanced Pid Control*. Research Triangle Park, NJ, USA: ISA, 2006.

[24] J. Kramer et al. “Self-Managed Systems: an Architectural Challenge”. In: *FOSE*. 2007, pp. 259 –268.

[25] L. Eeckhout et al. “Quantifying the Impact of Input Data Sets on Program Behavior and its Applications”. In: *J. Instruction-Level Parallelism* 5 (2003).

[26] N. Thomas et al. “A framework for adaptive algorithm selection in STAPL”. In: *PPoPP*. 2005, pp. 277–288.

[27] J. Ansel et al. “PetaBricks: A Language and Compiler for Algorithmic Choice”. In: *ACM PLDI*. 2009.

[28] W. Baek et al. “Green: A Framework for Supporting Energy-Conscious Programming using Controlled Approximation”. In: *ACM PLDI*. 2010.

[29] Y. Dotsenko et al. “Auto-tuning of fast fourier transform on graphics processors”. In: *PPoPP*. 2011, pp. 257–266.

[30] S. F. Rahman et al. “Automated empirical tuning of scientific codes for performance and power consumption”. In: *HiPEAC*. 2011, pp. 107–116.

[31] O. Krieger et al. “K42: Building a Complete Operating System”. In: *EuroSys*. 2006.

[32] Oracle Corp. *Automatic Workload Repository in Oracle Database 10g*. `http://www.oracle-base.com/articles/10g/AutomaticWorkloadRepository10g.php`.

[33] Intel Inc. *Reliability, Availability, and Serviceability for the Always-on Enterprise*. `www.intel.com/assets/pdf/whitepaper/ras.pdf`. 2005.

[34] M. Kihl et al. “Control-Theoretic Analysis of Admission Control Mechanisms for Web Server Systems”. In: *The World Wide Web Journal* 11 (2007), pp. 93–116.

[35] C. Lu et al. “Feedback Control Architecture and Design Methodology for Service Delay Guarantees in Web Servers”. In: *Parallel and Distributed Systems, IEEE Transactions on* 17.9 (2006), pp. 1014–1027.

[36] X. Dutreilh et al. “From Data Center Resource Allocation to Control Theory and Back”. In: *CLOUD* 0 (2010), pp. 410–417.

[37] D. Kusic et al. “Risk-aware limited lookahead control for dynamic resource provisioning in enterprise computing systems”. In: *Cluster Computing* 10 (4 2007), pp. 395–408.

[38] C. Cascaval et al. “Performance and environment monitoring for continuous program optimization”. In: *IBM J. Res. Dev.* 50.2/3 (2006), pp. 239–248.

[39] C. Karamanolis et al. “Designing controllable computer systems”. In: *HotOS*. 2005, pp. 9–15.

[40] M. Maggio et al. “Controlling software applications via resource allocation within the heartbeats framework”. In: *CDC*. 2010, pp. 3736–3741.

[41] H. Hoffmann et al. “Self-aware computing in the Angstrom processor”. In: *DAC*. 2012.

[42] R. Dorf et al. *Modern control systems*. Prentice Hall, 2008.

[43] X. Zhu et al. “What does control theory bring to systems research?” In: *SIGOPS Oper. Syst. Rev.* 43 (1 2009), pp. 62–69.

[44] M. Tanelli et al. “LPV model identification for Power Management of Web service Systems”. In: *MSC*. 2008, pp. 1171–1176.

[45] A. Filieri et al. “Reliability-driven dynamic binding via feedback control”. In: SEAMS. 2012, pp. 43–52.

[46] Q. Sun et al. “LPV Model and Its Application in Web Server Performance Control”. In: *CSSE*. Vol. 3. 2008, pp. 486–489.

[47] J. L. Hellerstein et al. “Applying control theory in the real world: experience with building a controller for the .NET thread pool”. In: *SIGMETRICS Perf. Ev. Rev.* 37.3 (2010), pp. 38–42.

[48] A. Leva et al. *Control-based Operating System Design*. Institution of Engineering and Technology, 2013.

[49] D. Perez-Palacin et al. “On the relationships between QoS and software adaptability at the architectural level”. In: *SoSyM Journal* 87 (2014), pp. 1–17.

[50] R. Calinescu et al. “Self-adaptive software needs quantitative verification at runtime”. In: *Commun. ACM* 55.9 (2012), pp. 69–77.

[51] V. Cardellini et al. “MOSES: A Framework for QoS Driven Runtime Adaptation of Service-Oriented Systems”. In: *IEEE Trans. Software Eng.* 38.5 (2012), pp. 1138–1159.

[52] S. Kounev et al. “Towards Self-Aware Performance and Resource Management in Modern Service-Oriented Systems”. In: *SCC 2010*, pp. 621–624.

[53] T. Zheng et al. “Integrated estimation and tracking of performance model parameters with autoregressive trends”. In: *ICPE*. 2011, pp. 157–166.