What Can Control Theory Teach Us About Assurances in Self-Adaptive Software Systems?

Marin Litoiu[1], Mary Shaw[2], Gabriel Tamura[3], Norha M. Villegas[3(✉)], Hausi A. Müller[4], Holger Giese[5], Romain Rouvoy[6,7], and Eric Rutten[7]

[1] York University, Toronto, Canada
mlitoiu@yorku.ca

[2] Carnegie Mellon University, Pittsburgh, USA
mary.shaw@cs.cmu.edu

[3] Universidad Icesi, Cali, Colombia
{gtamura,nvillega}@icesi.edu.co

[4] University of Victoria, Victoria, Canada
hausi@cs.uvic.ca

[5] Hasso Plattner Institute for Software Systems Engineering, Potsdam, Germany
holger.giese@hpi.uni-potsdam.de

[6] University of Lille 1, Villeneuve-d'Ascq, France
romain.rouvoy@univ-lille1.fr

[7] Inria, Saclay, France
eric.rutten@inria.fr

Abstract. Self-adaptive software (SAS) systems monitor their own behavior and autonomously make dynamic adjustments to maintain desired properties in response to changes in the systems' operational contexts. Control theory provides verifiable feedback models to realize this kind of autonomous control for a broad class of systems for which precise quantitative or logical discrete models can be defined. Recent MAPE-K models, along with variants such as the hierarchical ACRA, address a broader range of tasks. However, they do not provide the inherent assurance mechanisms that control theory does, as they do not explicitly identify and establish the properties that reliable controllers should have. These properties, in general, result not from the abstract models, but from the specifics of control strategies, which are precisely what these models fail to analyze. We show that, even for systems too complex for direct application of classical control theory, the abstractions of control theory provide design guidance that identifies important control characteristics and raises critical design issues about the details of the strategy that determine the controllability of the resulting systems. This in turn enables careful reasoning about whether the control characteristics are in fact achieved. In this chapter we examine the control theory approach, explain several control strategies illustrated with examples from both domains, classical control theory and SAS, and show how the issues addressed by these strategies can and should be seriously considered for the *assurance* of self-adaptive software systems. From this examination we distill challenges for developing principles that may serve as the basis of a control theory for the assurance of self-adaptive software systems.

© Springer International Publishing AG 2017

R. de Lemos et al. (Eds.): Self-Adaptive Systems III, LNCS 9640, pp. 90–134, 2017.
https://doi.org/10.1007/978-3-319-74183-3_4

1 Introduction

This chapter explores the contributions that classical feedback loops, defined by control theory, can make to self-adaptive software (SAS) systems, particularly to their assurances. To do this, we focus on the abstract character of classical feedback loops—how they are formulated, the assurances they provide, and the analysis required to obtain those assurances. Feedback loops, in one form or another, have been adopted as cornerstones of software-intensive self-adaptive systems, as documented in [33] and later on in [12,38]. Building on this, we study the relationships between feedback loops and the types of assurance they can help provide to SAS systems. We focus particularly on the conceptual rather than implementation level of the feedback model. That is, we concentrate on the relationships among desired properties that can be ensured by control theory applied in feedback loops (*e.g.*, stability, accuracy, settling time, efficient resource use), the ways computational processes can be adjusted, the choice of the control strategy, and the quality attributes (*e.g.*, responsiveness, latency, reliability) of the resulting system.

More concretely, this exploration includes, on the one hand, how feedback loops contribute to providing assurances about the behavior of the controlled system and, on the other hand, how assurances improve the realization of feedback loops in SAS. To set the stage for the discussion of concrete challenges identified in this exploration, we first review the major concepts of traditional control theory and engineering, then study the parallels between control theory [5,43] and the more recent research on feedback control of software systems (*i.e.*, MAPE-K loops) [29,33] in the realm of SAS. We establish a common vocabulary of key concepts, such as the control objective or setpoint, the disturbances that may affect the system, control responsiveness, and the control actions used by the controller to adapt the system. By discussing several types of control strategies, backed up by concrete examples, we illustrate how the control theory approach can direct attention to decisions that are often neglected in the engineering of SAS [12]. From the analysis of these concrete examples, we posit key challenges for assurance research related to the application of feedback loops in self-adaptive software.

The contribution of this chapter is twofold. First, it provides a comprehensive, and easy to understand, explanation of the application of control theory concepts to the engineering of SAS systems. This is valuable for both control engineering researchers interested in applying control engineering in new domains, and SAS researchers and engineers interested in taking advantage of control theory concepts and techniques for the engineering of SAS systems. Second, the chapter discusses assurance challenges faced by SAS designers and extrapolates control theory questions, concepts, and desirable properties into the SAS systems realm.

The remaining sections of this chapter are organized as follows. Section 2 revisits SAS foundational concepts, particularly the MAPE-K loop. Section 3 explains control theory concepts, including their analogy with the corresponding elements in SAS systems. Section 4 extends these models to adaptive and hierarchical control. Section 5 presents an overview of the application of control theory to the self-adaptation of software systems, focusing on the model and controller definitions. Section 6 discusses classic control strategies and the assurances they provide about the quality of the control, based on control theory properties; it also analyzes the design of controllers with desirable control theory properties. Section 7 discusses relevant assurance challenges in the design of SAS systems, extrapolated from those of control theory. Finally, Sect. 8 summarizes and concludes the chapter.

2 Self-Adaptive Software (SAS) Systems

The necessity of a change of perspective in software engineering has been discussed during the last decade by several researchers and practitioners in different software application domains [16,18,44]. In particular, Truex *et al.* posited that software engineering has been based on the incorrect assumption that software systems should support rigid and immutable business structures and requirements, have low maintenance, and fully satisfy their requirements from the initial system delivery [42,57]. In contrast to this "stable" vision, the engineering of self-adaptive software (SAS) originates from a different paradigm based on continuous analysis, dynamic requirements negotiation and incomplete requirements specification.

SAS systems adjust their own structure or behavior at runtime to regulate the satisfaction of functional, behavioral and non-functional requirements that vary over time, for instance when affected by changes in business goals or the system's execution environment [16,36,37,49,55]. When system requirements evolve in this way, the system must support both short-term adaptation mechanisms to ensure satisfaction of current requirements and also long-term evolution of the requirements [42,45]. The former requires continuous adjustments of system parameters to satisfy current requirements, and the latter requires runtime system analysis to direct long-term evolution [12,16]. Furthermore, assurance mechanisms, both at design time and runtime, must be implemented to guarantee required properties [56].

Inspired by the human autonomous nervous system, IBM researchers proposed the *autonomic element* as a building block for developing self-managing and autonomic software systems. They synthesized this adaptation in the form of the so-called Monitoring-Analysis-Planning-Execution and shared Knowledge (MAPE-K) loop model, as depicted in Fig. 1. The purpose of this model is to develop autonomous control mechanisms to regulate the satisfaction of dynamic requirements, specifically in software systems [29,30,33]. An autonomic manager not only implements the phases of the adaptation process, but also provides the interfaces (*i.e.*, manageability endpoints) required to sense the environment, and to effect changes on the target system (*i.e.*, the system to be adapted, also referred as the managed application) or to manage other autonomic managers.

The responsibilities of each phase of the MAPE-K loop depicted in Fig. 1 are defined as follows:

– *Monitoring.* In this phase sensors keep track of changes in the managed application's internal variables corresponding to desired properties (*e.g.*, measured

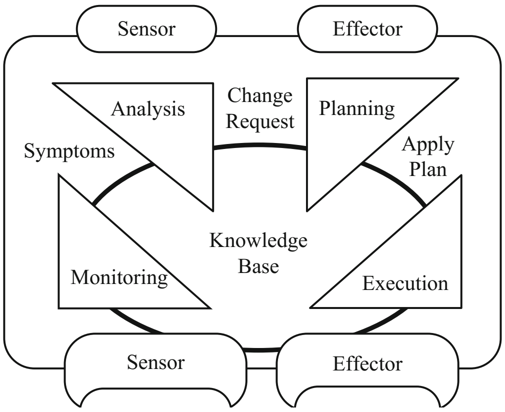

Fig. 1. The MAPE-K loop [33]

QoS data), the properties to be assured on the adaptation mechanism (*e.g.*, stability), and the external context (*i.e.*, measured from outside the managed application). Based on these changes, monitors must notify relevant context events to the analyzer. Relevant context events are those derivable from system requirements (*e.g.*, from QoS contracts). At this phase, the *knowledge base* can provide information, such as the values desired for the properties that are being controlled, and historical information that is useful for inferring context events.

– *Analysis.* Based on the high-level requirements to fulfill, the context events notified by monitors from the monitoring phase, and using the *knowledge base*, in this phase the analyzer determines whether a system adaptation must be triggered and notifies the planner accordingly. This would occur, for instance, when the notified events signal changes that (may) violate the reference control inputs.
– *Planning.* In this phase, once notified with a reconfiguration event from the context analyzer, the planner generates a strategy to fulfill the new requirements, using the accumulated knowledge in the *knowledge base*. By applying the generated strategy, the planner computes the necessary control actions to be instrumented in the managed software system as discrete operations that are then interpreted by the executor in the next phase.
– *Execution.* In this phase, after receiving a reconfiguration plan, the executor interprets each of the discrete operations specified in the plan and effects them on the managed software system. This implies translating or tailoring the reconfiguration actions to the ones supported by the platform that executes the managed software system. The *knowledge base* may provide information useful for the adaptation process, such as reconfiguration rules or runtime reference models.

3 Feedback Control

Control theory depends on reference control points of system behavior and corresponding explicit mathematical model specifications. Some researchers advocate the idea that control theory is applicable as is to SAS systems and propose a process for realizing this idea [25]. Basically, the idea is to find the right mathematical model for the software system behavior and apply the corresponding treatment depending on whether the model is linear or non-linear. This direct application of the methods of classical process control might be tractable for simple cases, but most practical self-adaptive systems are too complex to apply these methods directly. The approach of [25] only considers the case where process control applies directly.

Here, we take a more nuanced position—we posit that understanding the basics of classical control theory provides a foundation for a more abstract view, based upon which important design guidelines, conceived especially for controlling software systems, can be generated and that might otherwise be overlooked. That is, we argue that control theory provides a point of view that enables a designer to design more complex self adaptive systems more effectively. The control-based method brings separation of concerns between, on the one hand, the design of the process or system to be controlled, and, on the other hand, the controller for a particular adaptation strategy (as distinguished from ad-hoc approaches where both are mixed, making them more difficult to design and reuse). It also brings assurances offered by the model-based approach, even if the software systems as object of control are quite different from the classical target of control theory.

This section follows [31] and introduces the concepts and vocabulary of control theory. In particular, it discusses factors that affect the quality of control that can be achieved. At each stage, it discusses how control algorithms affect the behavior of systems and it identifies more abstract design tasks that are extensions of the basic theory. We focus here on discretized control, that is, on models with discrete time steps.

In a simple feedback system, a process P has a tuning parameter u (*e.g.*, a knob, which is represented by the little squared box in Fig. 2) that can be manipulated by a controller C to change the behavior of the process, and a tracked metric y that can be sensed in some way. The system is expected to maintain the tracked metric y at a reference level y_r, as illustrated in Fig. 2. At each time increment of the system, C compares the value of y (subject to possible signal translation) to the desired value y_r. The difference is the tracking or control error, and if they are sufficiently different, the controller changes parameter u to drive the process in such a way as to reduce the tracking error.[1] Somewhat confusingly, the tuning parameter u is often called the "input" and the tracked metric y is often called the "output." It is important to remember that these are the "control" input and output, not the input and output of whatever computation P is performing.

[1] If appropriate, the process P can be described by a model and the controller C might use that model to determine changes of u.

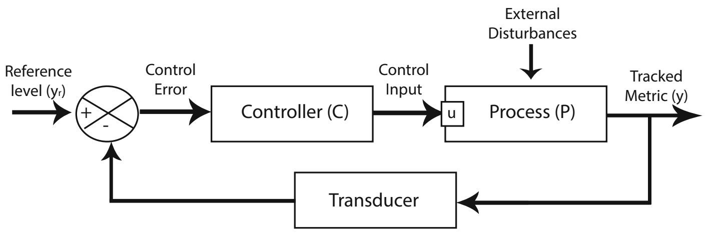

Fig. 2. The feedback loop control model.

For instance, assume the process P is a household thermostat to keep a room warm at a reference level of $y_r = 23\,^{\circ}\mathrm{C}$, and the controller C is a rudimentary *on/off* decision mechanism. Correspondingly, the tuning parameter u in P is a simple switch *on/off*. The tracked metric y is the temperature measured from a strategically located sensor in the room. Regularly, C compares the current value of y to the desired value y_r. If y is less than y_r, C makes u to be in the *on* position. Of course, in a next cycle, whenever y is greater or equal than y_r, C makes u to turn *off*, and so on.

Control systems are especially important when processes are subject to unpredictable disturbances, illustrated in Fig. 2 as “External Disturbances.” External disturbances are inputs into the Process P that affect the controlled output and they cannot be controlled. For our household thermostat example, external disturbances are the external variations in temperature, together with the room temperature changes created by the sudden opening of the room door and windows. In computing environments, external disturbances examples include, among others, variable loads, such as number of users or request arrival rates, and variable hit rates on caches. Control systems can also be important when the computation in the process itself is unpredictable and its accuracy or performance can be tuned by adjusting its parameters.

Control systems can also concern events and state based aspects, where we have not only discretized time as before, but also a discrete state space, which can be infinite, or finite but non-trivial because of the size and/or transitions. There exist different approaches to discrete control in self-adaptive systems: some are related to planning techniques from Artificial Intelligence or reactive synthesis in Formal Methods or Game Theory. An approach stemming from the Control Theory community is the supervisory control of Discrete Event Systems (DES).

3.1 Correspondences Between Feedback Control and MAPE-K

An important difference between the feedback loop from control theory and the MAPE-K loop is the complexity of each constituent component and of the overall loop. For example, in the former, the control actions are atomic operations for physical actuators (*e.g.*, resistors and motors) with clear causality between inputs and outputs, whereas in the latter, they are sequences of discrete plans (thus called reconfiguration plans) with long lags and uncertain consequences.

However, the basic principles are the same and, in the SAS domain, it is useful to identify the analogues of the control elements, ensuring that they can be used effectively for control. For example:

– What corresponds to the tracked metric (or output)? That is, what characteristics of the process are we trying to control? Can we measure them directly? If not, how are we going to infer their values? For example, is there a proxy value readily available? How accurately does the proxy value reflect the true value?
– What corresponds to the reference level? That is, how do we characterize the desired behavior of the system? Is it a specific target? Is it a limit value, so that y should always be less than the limit (*e.g.*, the temperature should approach the boiling point in a cooking pot but never reach it), or are values on either side of the target acceptable (*e.g.*, the cooking pot temperature should be 75 °C)? Is it a function of something else? If it is a complex function of other system parameters, how do we analyze interactions between the control discipline and the inputs to the function?
– What corresponds to the tuning parameters of the process and the way of modifying them? That is, in what ways can the controller modify the behavior of the process? Are these ways sufficient to maintain control of the process? (Old-style automotive cruise control systems failed this test, because they only control the accelerator, so they could only speed up the car, not slow it down).
– What effect a given change will have in the control input have on the process? In control theory, this is called the system identification problem. The more precisely we know this effect, the better our ability to achieve the response we want. For complex SAS systems, a full specification is often impractical. If so, it is more important to know whether increasing u will increase or decrease y, than it is to know exactly what the size of the effect will be.
– What are the sources of the external disturbances? What is their nature and how do we characterize them? What is their frequency and amplitude? In control theory, feedback loops are designed also to compensate for the effect of disturbances, within some bounds in amplitude and frequency. Can we characterize those bounds for SAS?
– How does the controller decide how much to change the control input variables? Control theory offers a rich space of control disciplines with well-understood effects on the controlled system, as explained later in Sect. 6.1. What are the analogous theories for SAS systems?

Figure 3 depicts structural correspondences between the feedback loop and the MAPE-K loop (Autonomic Manager). The *reference level* (y_r) of desired properties is fed by the system administrator to the analysis phase of the MAPE-K loop. In this phase, the analyzer evaluates the difference between the *tracked metric* y and the reference level, which is captured by the sensors of the Autonomic Manager. The analysis phase communicates the adaptation decision to the planning phase by sending a change request that corresponds to the *control error*.

The planning and execution phases correspond to the *controller*, which generates an adaptation plan and executes the adaptation of the managed system, *the process*, through the effector using the *control input* u (*i.e.*, the adaptation commands). Finally, sensors may play the role of the *transducer*.

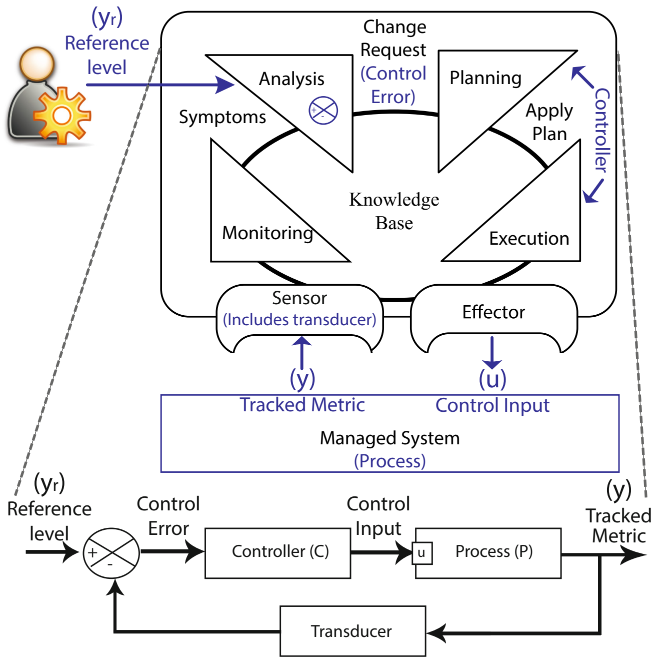

Fig. 3. Correspondences between the feedback loop and the MAPE-K Loop

The application of control theory to the engineering of self-adaptive applications has been actively studied by researchers in the SAS realm [12,16,38]. In particular, recent works have studied the usage of control theory to guide the design of MAPE-type systems, in different aspects of self-adaptation from the design of the adaptation mechanism to the assurance of its properties [56,58,60,61,63] To continue advancing in this direction, an important step forward is to support reasoning about the systems by finding the answers to the questions listed above about the correspondences between the feedback loop model from control theory and the MAPE-K loop model from SAS. These questions apply to a wide range of SAS systems, not simply to low-level control systems.

4 Adaptive and Hierarchical Control

For systems that vary in time or face a wide range of external disturbances, it is basically impossible to design one controller that addresses all those changes.

For these cases, we need to design a set of controllers or a parameterized controller. When the current controller is not efficient anymore, we change it or retune it. When the controller is updated while the system runs, we call it adaptive control. Adaptive control requires additional logic that monitors the efficiency of the controller under given conditions and, when some conditions are met, retunes the controller to adapt it to the new situation. The control community has dealt with adaptive control for decades, since 1961 according to Åström [5]. Åström provides a good working definition for adaptive control: "An adaptive controller is a controller with adjustable parameters and a mechanism for adjusting the parameters." This definition implies two (or more) layered control loops. Regular home thermostats are simple instances of on-off control that have no information about environment other than temperature. An hypothetical "adaptive thermostat" would use additional environment variables as well, such as humidity, door or window open. The values of these additional parameters (through some tuning parameters) would change the hypothetical thermostat control strategy (on-off switch times, for example). In the SAS area, the use of adaptive feedback loops was first suggested in [41]. One possible realization of an adaptive controller is depicted in Fig. 4. There are two feedback loops: the reactive feedback loop, at the bottom, that takes the output y and brings it to the comparator and further to the controller; and the adaptation loop at the top that estimates a dynamic set of models of the software (Model Estimators) passes the models (Models) to a Controller Tuning/Synthesizing component. This component uses a current Model to either recalculate the parameters of the controller or switch to a different controller.

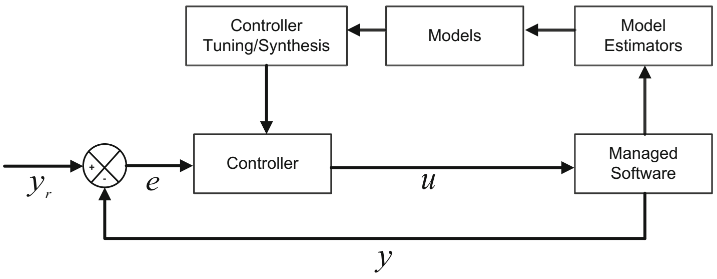

Fig. 4. Model Identification Adaptive Control

Hierarchical control, in addition to adaptive control, can be used for complex systems and complex controllers. The robotics community in particular has dealt with hierarchical control since the early eighties—they called it hierarchical intelligent control system (HICS). In the software engineering community, most notably Kephart and Chess [30,33] introduced autonomic systems and autonomic computing reference architecture (ACRA).

Concerning discrete control, notions of hierarchies have also been considered. One is the fact that there is often a need for coordination of, or switching between, different quantitative controllers, each with specific objectives. Between levels of discrete control, hierarchies can also be established as in *e.g.*, [19,21]. Adaptive discrete control, on the other hand, is a notion less developed than for quantitative control, and more theoretical work is needed.

The autonomic computing reference architecture (ACRA), depicted in Fig. 5, provides a reference architecture as a guide to organize and orchestrate SAS (*i.e.*, autonomic) systems using autonomic managers (MAPE-K loops). SAS systems based on ACRA are defined as a set of hierarchically structured controllers. Using ACRA, software policies and assurances can be implemented into layers where system administrator (manual manager) policies control lower level policies. System operators have access to all ACRA levels.

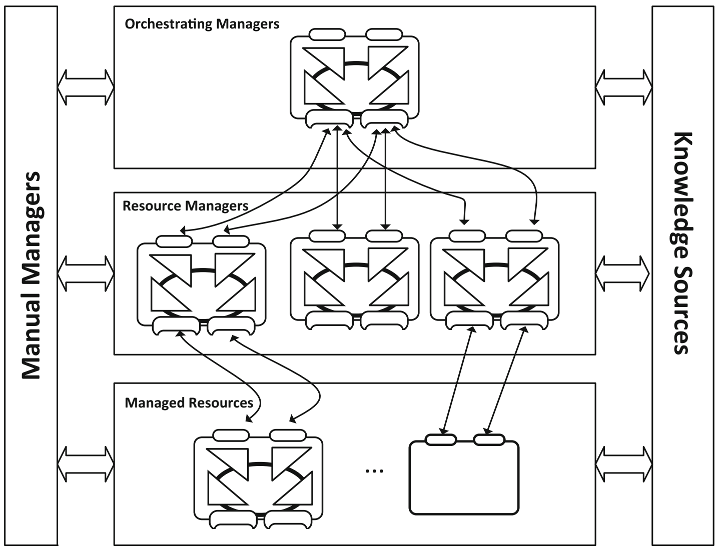

Fig. 5. The Autonomic Computing Reference Architecture (ACRA) [30]

Hierarchical control for SAS was further specialized by Kramer and Magee in a three layers architecture for SAS [34]. Each layer has a different time scope and authority. The lowest layer, *component control*, contains all the mechanisms to support changes in components; the middle layer, *change management*, contains the plans that are executed by the lowest layer; and the upper layer, *goal management*, produces change management plans as requested by the layer below and in response to the introduction of new goals.

5 Control Theory Applied to Self Adaptive Software - An Overview

Over the last decade, there have been many efforts to use control theory to design feedback loops for self-adaptive software systems. Hellerstein *et al.*, in their 2004 book [29], introduced several examples where control theory has been applied for controlling the threading level, memory allocation or buffer pool sharing in commercial products, such as IBM Lotus Notes and IBM DB2. Other researchers and practitioners have published results on control theory for computer power control [35], thread and web cluster management [1], admission control, video compression [24], performance and denial of service attack mitigation (*e.g.*, in the Apache server) [20], to name just a few examples.

In the examples we have seen so far, the authors use feedback loops to control quantitative metrics from the categories of performance, cost, energy, reliability. In these control examples, the methodology and the mathematical apparatus follow the control theory methodology and revolves initially around two basic concepts: the model of the controlled subsystem (*i.e.*, the open loop model from the point of view of the controlled metric), and the controller to close the loop, as follows.

5.1 The Open Loop Model

The first major task in applying control theory to SAS is to create a model for the quality attributes to be controlled. Since this model focuses only on the controlled or managed system (that is, the output is not considered to compute the control input), we call it open loop model. The model is quantitative and captures, in a very specific format, the relationships among software components, and between software and environment. The model can be constructed analytically by writing the equations of the modelled phenomena or by following an experimental methodology called system identification [39,51]. In general, defining the model starts by identifying:

– the outputs that need to be controlled, y;
– the disturbances, p, that affect the outputs;
– the control variables (or commands), u, that can control the outputs of the system and compensate for the effects of external disturbances; and
– the internal state variables, x (x can be seen as an intermediate link between u and y).

The external disturbances, p, drive the system towards undesired states. For example, an external load increase (increase in number of users) might increase the utilization (state) of a server and therefore increase the response time (output). Note that the perturbations p do not affect the response time directly, but through the intermediate state that we call utilization. To compensate the effects of the external load, a feedback loop designer should engineer a control input that changes the utilization of server in the opposite direction (like transferring some of the load to another server). Note that u, x, p, y are vectors, not scalars, which means that a system has many input, output, state or external disturbance variables. In control terminology, this is defined as MIMO (multiple input, multiple output) system.

In the general case, with f and g non-linear discrete *vector* functions, the open loop model can be described as:[2]

$$x(k+1) = f(x(k), u(k), p(k)) \tag{1}$$

$$y(k) = g(x(k), u(k)) \tag{2}$$

where k is the current discrete time and $k+1$ means the "next time increment." The magnitude of the difference between k and $k+1$ depends on exactly what the model tries to capture. Equation 1 projects the state forward, as a function of the current state x, some control commands u and disturbances/perturbations, while Eq. 2 expresses the output as a function of the same parameters.

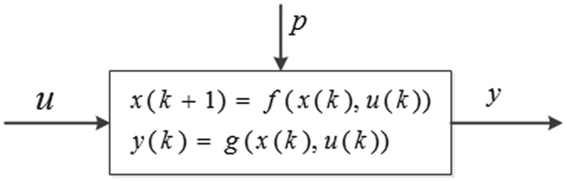

Fig. 6. A discrete non-linear model

The above model formulation, omitting the disturbance p as depicted in Fig. 6, is general enough to describe any quantitative aspect of a system that might need adaptation, including aspects of SAS. When constructing the above model for a software system, a special attention should be paid to identifying (or engineering) u, the control inputs or commands. Those are variables in the system that control engineers pay special attention to and they are a *sine qua non* condition for designing and engineering a feedback loop. The commands u are control knobs, that is software parameters, scripts, programs, interfaces through which a system administrator or an automation program can change a software configuration. They are always designed with some knowledge about the possible external disturbances. Load inputs to the systems, such as workloads, are frequently modeled as external disturbances or perturbations, p, because they are outside the control of the SAS designer. Their effects are compensated for by designing control inputs u, engineered into the system: for example by increasing the capacity of the server, scaling out by replicating servers and distributing the load, changing the data flows inside the system or changing the threading levels.

5.1.1 Continuous Linear Open Loop Models

In many cases, the non-linear model can be simplified or approximated with a linear one, such as the one below. In this model we assume an additive external perturbation:

[2] In many cases the models include explicitly the modelling and measurement noise; to keep the presentation simple, in this chapter we do not represent noise explicitly.

$$x(k+1) = A * x(k) + B * u(k) + F * p(k) \tag{3}$$

$$y(k) = C * x(k) + D * u(k) \tag{4}$$

where A, B, C, D, F are constant matrices that can be determined experimentally for a specific deployment and under particular perturbations.[3]

Example 1: Software queues modeling (performance view). Many software entities can be modelled as queues (semaphores, critical sections, thread and connection pool containers, web services, servers, etc.). Although the performance metrics of those entities can be captured with non-linear models, many authors prefer to linearize the models around an equilibrium point and express them with the canonical model (3) and (4). Below is an example of such model for a web server (taken from [20,29]). In Eqs. (5) and (6), x_1 and x_2 represent the server utilization and the memory usage, respectively; the commands, u_1 and u_2 represent the number of HTTP connections and the keep-alive intervals; and y represents the response time of the server. A_{ij}, B_{ij}, where $i, j = 1, 2$ and C_1 are constants that have to be determined experimentally for a specific server deployment. Intuitively, the model says that the future memory usage and the server utilizations are a function of the current usage, the number of connections, and the length of interval they are kept alive.

$$\begin{bmatrix} x_1(k+1) \\ x_2(k+1) \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} x_1(k) \\ x_2(k) \end{bmatrix} + \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} \begin{bmatrix} u_1(k) \\ u_2(k) \end{bmatrix} \tag{5}$$

$$y(k) = \begin{bmatrix} C_1 & 0 \end{bmatrix} \begin{bmatrix} x_1(k) \\ x_2(k) \end{bmatrix} \tag{6}$$

Similar models have been presented for various entities that contain queues [3,24,28,52,53]. They were derived from the dynamics of the queues and from experiments. For example, Solomon *et al.* [53] use a control theory specific model identification methodology to experimentally identify A, B, C, and D for a threading pool.

5.1.2 Discrete Open Loop Models

If in contrast a finite discrete model and discrete control is considered, we may use instead of a linear infinite model a discrete transition system. A simple case to describe the open loop model would be Moore automata $M = (X, \Sigma, \Omega, \delta, \mu)$ with state space X, input event set Σ, output event set Ω, total state transition function $\delta : \Sigma \times X \to X$, and total output function $\mu : X \to \Omega$. Assuming events sets U, P, and Y for u, p, and y, respectively, we could use a Moore automaton $M_{ss} = (X_{ss}, \Sigma_{ss}, \Omega_{ss}, \delta_{ss}, \mu_{ss}, x_{ss})$ with X_{ss} the possible states x, $\Sigma_{ss} = U \times P$, $\Omega_{ss} = Y$, and $x_{ss} \in X_{ss}$ the start state to describe the software system as follows:

$$x(k+1) = \delta_{ss}((u(k), p(k)), x(k)) \quad y(k) = \mu_{ss}(x(k)) \tag{7}$$

[3] Here we assume a time invariant case, but those values can depend on time as well.

As we chose $\Sigma_{ss} = U \times P$, the automaton adjusts its state in a single step according to the command u and the disturbance p. Therefore, a simple scheme might be that $U = \{pass, block\}$, such that for *pass* the software system operates as usual and for *block* the software system ignores the external input.

5.2 The Closed Loop Model

In the general case the feedback control can only observe the software system represented by the open loop model at its output y, while the disturbance p and state x are not directly observable.

5.2.1 Continuous Linear Closed Loop Models

For the linear case, Eqs. (3) and (4) suggest that if we want to keep y at a predefined value y_r (the goal), we accomplish that by computing a command u_r based on p that compensates the disturbance (feed-forward control). In practice, that is impossible for two main reasons: (a) the model (2) is an inaccurate representation of the real system; (b) there are external disturbances, p, that affect the system. A feedback loop implies adding a new software component, a controller (or Adaptive Manager) as depicted in Fig. 7, that is fed back with information from the controlled software system.

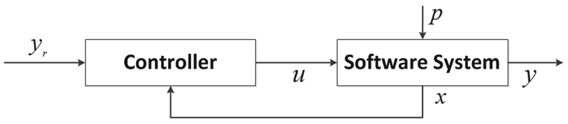

Fig. 7. Controller and feedback loop.

The controller and the feedback will compensate for modeling errors and for external disturbances. Furthermore, they can also stabilize unstable systems and achieve additional control criteria. The feedback can be taken from the state variables (state feedback) or from the output (output feedback). The state feedback is the most effective, since it assumes an understanding of inner workings and dependencies of the software system. Output feedback is more practical since output variables are easier to measure. For observable systems, a system for which its internal state variables can be estimated from measured variables, a state feedback can be constructed from output feedback, using estimators as for example Kalman filters [32]. When using estimators, we measure y; estimate x based on the model, the observability matrix and the measurements; and then compute a state based controller. The controller can vary from a simple constant matrix to a complex set of differential (or difference) equations and can be synthesized to achieve some design goal, such as stability of the entire system, perturbations rejection across wide ranges (robust control), or optimization. While the controller design goals can be diverse, there are standard design techniques that have well defined procedures and solutions.

The interesting aspect of the feedback controller is that the overall equations of the closed loop system (or closed loop model) can be inferred from the open loop model and the equations of the controller. For example, consider the controller a matrix K, that needs to be determined. If K is placed on the feedback loop, then $u(k) = y_r$-K*$x(k)$. The equations of the closed loop system, having y_r as input and y as output, can be computed by substituting u in Eqs. (2) and (3), and will look as follows (for simplicity, F = 0):

$$x(k+1) = A_c * x(k) + B_c * y_r(k) \tag{8}$$

$$y(k) = C_c * x(k) + D_c * y_r(k) \tag{9}$$

where $A_c = (A - BK)$; $B_c = B$; $C_c = (C - DK)$; $D_c = D$.

Having the equations of the closed loop system allows the designer to shape the behavior of the states and therefore of the outputs by choosing the right K. At the same time, a model will enable the analysis and verification that the system has indeed the desired behavior.

5.2.2 Discrete Closed Loop Models

For the case of discrete control, a Moore automaton $M_c = (X_c, \Sigma_c, \Omega_c, \delta_c, \mu_c, x_c)$ with X_c the possible states of the controller, $\Sigma_c = Y$, $\Omega_c = U$, and $x_c \in X_c$ the start state to describe the controller for the software system of Eq. 7, the resulting closed loop model is as follows for x' the state component of the controller[4] and x the state of the software system:

$$x'(k+1) = \delta_c(y(k), x'(k)) \tag{10}$$

$$x(k+1) = \delta_{ss}((\mu_c(x'(k)), p(k)), x(k)) \quad y(k) = \mu_{ss}(x(k)) \tag{11}$$

For the system described in Sect. 5.1.2, consider simple scheme with $U = \{pass, block\}$, such that for *pass* the software system operates as usual and for *block* the software system ignores the external input. A controller can thus always block an input if this would lead to an unsafe behavior. However, the controller has to deduce this from the observed outputs of the software system.

5.3 Feedback Control Behavior

The main challenge now is to chose the right strategy for the controller such that the closed loop model and its feedback control behavior has the required properties.

[4] To stay closer to the continuous case, we consider here the case of a discrete controller with an own state space distinct from the state space of the controller process that is more general than the supervisory control approach.

5.3.1 Continuous Linear Control

In the case of continuous control, independent of having a model for the open or closed loop, a controller for a SAS achieves its goals by successive approximations. It is hard to know exactly how a given system will respond to changes in the control inputs (*i.e.*, the system dynamics).

Control algorithms operate by making successive corrections that are designed to reduce the tracking error. Several factors lead to this strategy, all of them derived from the uncertainties about the exact system dynamics: delays in system response (*e.g.*, startup time to bring a new server online), lags in system response (*e.g.*, the time it takes to clear a backlog), environmental changes and events (*e.g.*, the execution of the garbage collector), and variability in the interaction of the controlled process with its environment (*e.g.*, randomness in arrival rates to a queue, or processing times for the jobs in a queue). Feedback loops provide some robustness in the face of these uncertainties. However, the consequence of their use is that the process approaches the reference level in a series of steps. For example, in the case of the home thermostat, the controller turns the furnace *on* and evaluates how the temperature goes with respect to the reference value. Depending on the results, the thermostat controller turns the furnace *off* and evaluates the results again. This process is continuously repeated to reach the desired temperature.

The design of the control algorithm determines whether the path to achieving the reference value is a series of overshoots above and below the desired value or a gradual approach to the reference value. The former strategy may get close to the reference value faster, but at the cost of oscillating behavior (imagine riding in a car that entered a new speed zone and made drastic changes up and down in speed, first overshooting and then undershooting to get to the new speed limit). The latter strategy may take longer to reach the reference value, but with much smoother behavior and less resource wasting. These are the aspects of concern when analyzing control properties and respective assurances.

Example 2: Balancing Job Throughput and Parallelism in Hadoop. In recent years, Hadoop has emerged as the *de facto* standard for big data processing and the MapReduce paradigm has been applied to a wide variety of applications and workloads, including distributed sorting, log analysis, document clustering and machine learning just to name a few. In this context, the performance and the resource consumption of Hadoop jobs do not only depend on the characteristics of applications and workloads, but also on an appropriately configured Hadoop environment. Next to the infrastructure-level configuration (*e.g.*, the number of nodes in a cluster), the Hadoop performance is affected by job- and system-level parameter settings. For example, YARN (*Yet Another Resource Negotiator*), the resource manager introduced in the new generation of Hadoop (version 2.0), defines a number of parameters that control how the MapReduce jobs are scheduled in a cluster, affecting the global jobs performance. Among these parameters, the MARP (*Maximum Application Master Resource in Percent*) property directly affects the level of MapReduce job parallelism and associated throughput. This parameter controls how the capacity scheduler balances between the number of concurrently executing MapReduce jobs and the number of map and reduce tasks. An inappropriate MARP configuration will therefore either reduce the number of jobs running in parallel resulting in idle jobs or reduce the number of map/reduce tasks and thus delaying the job completion. However, finding an appropriate MARP value is far from trivial. On the one hand, the diversity of MapReduce applications and workloads suggests that a simple, one-size-fits-all application-oblivious configuration will not be broadly effective—i.e., one MARP value that works well for one MapReduce application/workflow combination might not work for another. On the other hand, YARN configuration is static and as such it cannot reflect any changes in workload dynamics. In this context, we propose the design of a YARN controller that autonomously balances the throughput and the parallelism of Hadoop jobs in order to minimize their completion time and maximize the resource utilization of the Hadoop cluster. This control problem can therefore be modeled as follows (cf. Fig. 8).

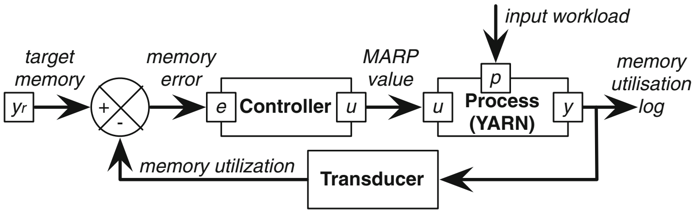

Fig. 8. Self-balancing controller for YARN.

To assess this controller, we executed all experiments on an Hadoop cluster with 11 physical hosts (1 control node and 10 processing nodes) deployed on the Grid5000 infrastructure.[5] We use SWIM (*Statistical Workload Injector for Mapreduce*) to generate 4 realistic MapReduce heterogeneous workloads. SWIM contains several large workloads (thousands of jobs), with complex data, arrival, and computation patterns that were synthesized from historical traces from Facebook 600-nodes. Figure 9 illustrates that one can observe for each workload that, compared to the vanilla configuration, our approach can significantly reduce the completion time of jobs (*e.g.*, up to 40% in W1). It also systematically delivers a better performance than the best-effort configurations.

5.3.2 Discrete Control

Discrete control focuses not on quantitative aspects of a system as shown before, but rather on logical, event-based abstractions. In addition to a discretized time coming with the notion of events, this form of control considers a discrete state space, which can be infinite, or finite but non-trivial because of its size and/or complexity of transitions.

[5] https://www.grid5000.fr.

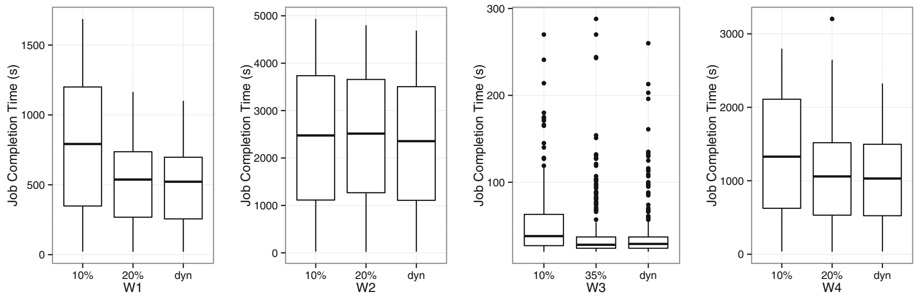

Fig. 9. The comparison of job absolute completion time observed for static and dynamic configuration parameters.

For the discrete control example with Moore automata $M_{ss} = (X_{ss}, \Sigma_{ss}, \Omega_{ss}, \delta_{ss}, \mu_{ss}, x_{ss})$ with X_{ss} the possible states x, $\Sigma_{ss} = U \times P$, $\Omega_{ss} = Y$, and $x_{ss} \in X_{ss}$ the start state to describe the software system and $M_c = (X_c, \Sigma_c, \Omega_c, \delta_c, \mu_c, x_c)$ with X_c the possible states of the controller, $\Sigma_c = Y$, $\Omega_c = U$, and $x_c \in X_c$ the start state to describe the controller for the software system combined in Eq. 10 with the simple blocking scheme supported by the control input u, the safety objective may be to restrict automata M_{ss} to the safe states $X_{ss}^{safe} \subset X_{ss}$. Then, the controller synthesis would synthesize a Moore automata M_c that blocks the Moore automata M_{ss} where necessary, such that for every sequence of external inputs p holds that M_{ss} never reaches an unsafe state in $X_{ss} \setminus X_{ss}^{safe}$. Maximally permissive here means that M_c use the *block* control input only to the absolut minimal extent that is necessary to exclude unsafe states.

There exist different approaches to discrete control in self-adaptive systems: some are related to planning techniques from artificial intelligence *e.g.*, [10,21,54], game theory or reactive synthesis in formal methods *e.g.*, [9,22]. Such approaches concern control objectives as expressive as of safety, reachability or even liveness. An approach stemming from the control theory community is the supervisory control of Discrete Event Systems (DES) [14,47,62]. Typical models are transition systems like Petri nets or automata, and the properties considered are logical and pertain to states and paths in the state space, like safety, reachability and non-blocking. For some classes of problems, there exist automated state-space exploration algorithms for Discrete Controller Synthesis (DCS), some of them implemented in efficient tools. One particularity is that, for safety objectives, the controller can be ensured to be maximally permissive (see also [48]).

In the next section, based on the model, the controller and the control behavior definitions, we analyze different control strategies and the assurances they provide with respect to control properties, the analysis of the open model properties, the design (or synthesis) of the controller, and then the analysis of the resulting closed loop model.

6 Assurances

Most physical processes, and many computational processes (especially complex ones), do not respond immediately to changes in their control inputs. Further, most systems do not respond so precisely that an exact correction can be made in a single step. Therefore, the design of a feedback system, whether it be a simple feedback loop or a complex SAS, must take into account the way the controlled process responds to changes in the control inputs. The characteristic responses constitute the basis on which system properties are built, and the assurances correspond to what (and to what extent) can be guaranteed about these properties.

This section focuses on *assurances* that can be realized with the help of feedback loops, and we explore their correspondences in SAS domains. In other words, we focus on what we can ascertain when using feedback loops, and explore the correspondences of these assurances in SAS domains. Of course, ideally, it would be desirable that just by using feedback loops we would have granted assurances on desirable properties and optimum behavior in the controlled software applications. For instance, for regulating the throughput of the web service for computing definite integrals it would be desirable to have a controller that always (i) corrects errors quickly; (ii) reaches the reference level with precision and minimum error; (iii) spends the resources at the minimum required; and (iv) reaches stability after achieving the reference level. However, obtaining such a controller for a SAS system in general is not easy and constitutes precisely the main challenge. Moreover, conflicts often arise when trying to achieve all of these control objectives together (*e.g.*, fast settling time usually imply significant overshoot), and thus, trade-offs must be considered between these objectives and the implied assurances.

In the following subsections, we discuss different control strategies and the way they can be exploited to assure properties and desirable behaviors. Then, we analyze the open loop model properties, the conditions that can help design controllers that provide assurances on desirable properties, and then the analysis of the resulting closed loop model properties. Even though we do not provide a complete answer to the main challenge, this analysis helps understand better the implied problems and sets the basis for defining the principles on which a software controlling theory should emerge.

6.1 Classic Control Strategies

From the discussions of previous sections and the questions posed in Sect. 3.1, we can infer that decisions about the control strategy are absolutely critical to the design of SAS. Classical control theory operates in settings that are simple enough that mathematical analysis is usually possible. SAS systems, on the other hand, are usually too complex to characterize completely. This makes the design and validation of the control strategy even more important for this kind of systems.[6]

This section reviews common control strategies and discusses their application to SAS. For this application, we use an extremely simplified version of the WolframAlpha web services[7] as the software to be controlled. Our web service only computes the integral of elementary functions (*i.e.*, functions with exponentials, logarithms, radicals, trigonometric functions, and the four basic arithmetic operations).

The service is implemented using an exact method for computing definite integrals of a given elementary function, that is, by computing the symbolic integration of the function and evaluating the result on the integration limits. This method is slow (*i.e.*, worst case is exponential, even though for many cases it is polynomial [11]), but exact.

6.1.1 On/Off Control

The simplest type of control is for a system where the control inputs of the managed system are either turned *on* or *off*. The most common example is the old-style household thermostat, where the thermostat turns the furnace *on* or *off*, as described previously (cf. Sect. 3). This simple strategy has significant drawbacks: it oscillates constantly (and hence is often called "bang-bang" control). This problem can be moderated by introducing hysteresis, that is, by delaying the control response until a modest overshoot has occurred.

To illustrate the application of the *on/off* control to our web service for computing definite integrals, assume we deploy the control method implementation both on a main server and on a spare server. Assume also that the required reference level is to maintain a given throughput of definite integral computation requests per time unit. Whenever the main server can service the requests maintaining the required throughput, the controller makes u to turn *off* the spare server, such that the spare server does not accept any new requests and shuts down upon finishing any pending requests. Otherwise, the controller makes u to turn the spare server *on* and redirects requests to it. However, independent of whether the number of requests grows in greater magnitudes, the action of this type of controller is clearly limited to activating the spare server.

Similarly, for the hadoop example we can assume a predefined set of spare processing nodes[8] that can be switched on and off depending on the requests throughput.

[6] We often describe "self-adaptation" in terms of strategic changes to maintain the system behavior as close as possible to the reference value. Note that a system can also self-adapt by changing the control discipline (adaptive control). This is another reason to make explicit design decisions about the control strategy.

[7] http://www.wolframalpha.com

[8] Hadoop does not support the addition of processing nodes that have not been considered during initial deployment.

6.1.2 Proportional Control

This type of control acts by making the size of the adjustment on the control input proportional to the size of the tracking error. In the simple case, the constant of proportionality is called the gain. The proportional gain determines the ratio of output response to the tracking error. For instance, if the error term has a magnitude of 10, a proportional gain of 2 would produce a proportional response of 20. In general, increasing the proportional gain will increase the speed of the control system response. Even though this strategy means a fast response to eliminate the tracking error, a shortcoming of proportional control is that it generally is unable to eliminate the tracking error entirely, a phenomenon called proportional droop. Thus, one critical design task is gain estimation. Gains that are too high result in excessive oscillation, possibly never reaching convergence toward the reference level, while gains that are too low result in sluggish performance. The effect of larger and smaller gains is further illustrated in Sect. 6.5.3.

Revisiting the thermostat example, modern high-efficiency furnaces run at several power levels:[9] the level 1 is energy-efficient but produces lower temperatures (*i.e.*, has lower gain), and the others gradually produce more heat at the expense of efficiency (*i.e.*, have higher gains). For an automated furnace with eleven internal power levels (0 being the Off position, 10 the maximum power) covering a range between 18 °C and 27 °C, the thermostat's ordinary operation uses the energy-efficient level in most cases, but at the expense of slower adjustment of temperatures in the living area. Assuming the thermostat has a proportional gain of 3, and the temperature error is of $3°C$, the controller would produce a command control for selecting the ninth power level position in order to correct the temperature rapidly; notice that since this is almost the maximum power level, maintaining it for too long would lead to significant overshoots, and therefore, correspondingly large oscillations.

For the application of this type of control to our web service, assume we deploy the *implementing* method on the servers of a cluster computing infrastructure, and that the required reference level is the same throughput as the defined for the *On/Off* control example. In this case, we estimate the gain as 2/100 times the tracking error. That is, in short, and roughly speaking, the resulting command control on this strategy is $u(k+1) = \frac{2}{100} * e(k)$, being $e(k) = y(k) - y_r(k)$. Thus, if the error reaches a difference of 200 in the throughput, the controller response is to activate four spare servers and distribute the service requests among them with the purpose of maintaining the required throughput. Nonetheless, despite the fast corrective action, small tracking errors (*i.e.*, below 50 in this case) produce no correction (*i.e.*, evidencing proportional droop).

Similarly, for the hadoop cluster, we can implement the MARP parameter tuning with this type of control by adjusting the percentage of resource allocated to application master, according to the current memory consumption.

[9] https://www.ecobee.com

6.1.3 Integral Control

As analyzed in the proportional control, independent small tracking errors can produce null corrective actions. However, the accumulation of these errors sooner or later should produce a corrective action, possibly small, but significant enough. The accumulated errors are essentially the integral of the error (or sum for discrete systems), so this is called integral control. However, given that this strategy applied alone naturally results in very slow responses, a common control strategy is to combine an integral control term with a proportional one. This combination provides a fast response action, with elimination of remnant small tracking errors.

For the thermostat example, assume the reference level is set to $25°C$, the controller having an integral coefficient of 3/4, and the current tracked metric being $24.5°C$. Having an integral term alone would yield a command control of $0.5 * 3/4 = 0.375$, that is, selecting the *Off* position for the furnace. After 3 more cycles with the same error (*i.e.*, with null corrective actions) a first corrective action is produced by setting the furnace on the first position ($0.375 * 3 = 1.125$). However, as this power level setting is too low to achieve and maintain $25°C$, the controller will have to keep accumulating errors until reaching the eighth power level, in which, after some more cycles, it will stabilize (recall from the initial specifications that the tenth power level achieves $27°C$. Once achieved the reference level, the temperature error will be diminished to near zero, contributing in this amount to the integral term. Indeed, the main contribution of the integral term, besides eliminating remnant tracking errors, is to maintain achieved levels of control (*i.e.*, the eighth power level) at a given point, in which other control terms (*e.g.*, the proportional one) would be null. Thus, this term models inertial load effects that are inherent in physical systems.

For the application of the integral control to our web service and hadoop case studies, assume the same deployment and required reference level as in the corresponding proportional control example. Also, let us define the integral control term as $I(k+1) = I(k) + \frac{1}{5} * e(k)$, that is, $u(k+1) = I(k+1) = \sum_{j=1}^{k} \frac{1}{5} * e(j)$. Thus, an independent small tracking error $e(k)$ of 2 units produces no immediate corrective actions. Nonetheless, if this error is maintained, after three control cycles the accumulated error is of 6 units, and the integral control term reaches 6/5. Then, the controller response is to activate one ($6/5 = 1.2$) spare server and distribute the service requests.

6.1.4 Derivative Control

It is important in many cases to predict or anticipate the software behavior trends. If we would know that in the near future the tracking error is going to increase, then we would take a proactive action now. In many cases, performing the command u, which in many cases is a work-flow, might take several minutes. This is called an execution lag. In these cases, the command will affect the output y not instantaneously, but with a lag of several minutes. It is better in these situation to anticipate the *trend*. The simplest way to accomplish it is to determine the sign and magnitude of the error derivative (or difference for discrete systems, $e(k) - e(k-1)$), and calculate u as a function of this difference.

Applied to the thermostat example, assume the same reference level of $25°C$, the controller having a derivative coefficient of 3/4, and the current tracked metric being $24.5°C$. This derivative controller will not react until the error *trend* becomes significant, given that the derivative term alone yields a command control of $(e(k) - e(k-1)) * 3/4$. When the error difference reaches 4/3, the controller produces the first corrective action by setting the furnace in the first position $((4/3) * (3/4) = 1)$, which, depending on the control cycle duration, will correct the error trend the more or the less. Even though this power level can eliminate the error *trend*, in absence of other control terms it is too low to achieve and maintain $25°C$. Moreover, as this controller reacts to error differences, trying to nullify the error trend, it alone will take a long time to reach the reference level set.

For the application of this type of control to our web service, assume the same deployment and required reference level as in the corresponding proportional control example. Also, let us define the derivative control command as $u(k+1) = \frac{1}{100} * (e(k) - e(k-1))$. Notice that, given that this strategy is based on the error trend, it should be used to complement other strategy: even if the error is large, but the trend is negative (*e.g.*, $e(k) - e(k-1) = -100$) indicating that the error is diminishing, the controller response is to deactivate one spare server. However, if we combine it with the proportional strategy in our example, the command control would be $u(k+1) = \frac{2}{100} * e(k) + \frac{1}{100} * (e(k) - e(k-1))$. In this case, if the error presents a difference of 200 in the throughput, but the error difference trend is -100, the corrective action is to activate 3 spare servers (not 4, as the proportional control alone would yield in this case).

Example 3: A PID Controller for Cluster Utilization. Figure 10 shows a PID controller presented by Gergin *et al.* [26] for the control of a software cluster. The controlled variable is the cluster utilization, y, and the command, u, is the number of software replicas within the cluster. A PID controller computes the number of replicas u as a function of error $e = y - y_r$, where y_r is the setpoint utilization for the cluster. The PID controller amplifies the error (K_p term), integrates all past errors (K_i term) and anticipate the direction of the error (K_d term). The coefficients K_p, K_i, K_d are determined experimentally or based on the system model in such a way that some quality control metrics (overshoot, rising and settling time, steady errors) are achieved.

6.1.5 Discrete Control

When the properties or characteristics which are to be controlled in a self-adaptive system concern logical or qualitative aspects rather than quantitative aspects, then a different type of control techniques has to be applied. Discrete control exists under different forms, for example planning techniques from Artificial Intelligence have been used to synthesize the sequences of low-level actions necessary to reconfigure a system according to high-level goals, or in more ambitious multi-level

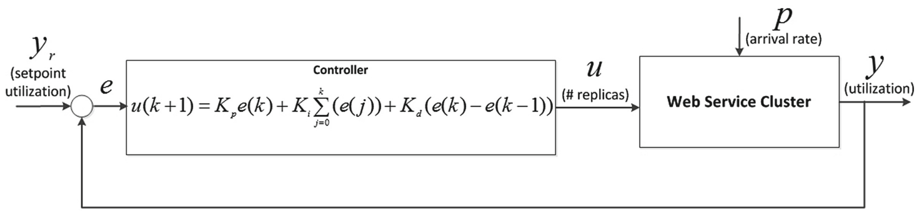

Fig. 10. A feedback loop with a PID controller for distributing the web service load in a cluster.

architectures to cope with unpredicted situations [21]. A form of discrete control, usually applied to flexible manufacturing, is the supervisory control of Discrete Event Systems (DES) [14]: their characterization is given by the nature of the state space of the considered system: when it can be described by a set of discrete values, like integers, or vectors of booleans, and state changes are observed only at discrete points in time, then such transitions between states are associated with events. These supervisory control techniques are beginning to be used for self-adaptive computing systems [48,62].

6.2 Control Theory to the Rescue of the MAPE-K Loop

From the above discussion, it is apparent that each control strategy poses different advantages and disadvantages (*i.e.*, implying different properties). Moreover, some of the strategies can be combined, as in Example 2, being it usual to search for combinations that exploit the advantages of ones to compensate the disadvantages of others, given a particular control problem.

In addition to the ones discussed, there are many other control disciplines and strategies. However, the presented examples should be sufficient to show how critical it is to design the control strategy of a SAS system very carefully. It is important to notice that no other software engineering discipline forces attention to these matters. For instance, the well-known MAPE-K model, proposed for implementing autonomic computing and self-management computing systems, does not forcefully raise these design issues. Of course, the design of the planning element of the model should address these matters, but its specification is purely functional, being concerned fundamentally with *what* is required in order to modify the system behavior. Our discussion argues for balancing the concerns, considering even more importantly the longer term effects of the proposed behavior modifications, that is, the *properties* implied by the chosen control strategy, over the specific technique or method for producing the modification.

Any discussion about the properties of the closed loop system should start with a discussion about the properties of the open loop system. Only certain properties of the open loop system will allow us to design and analyze the closed loop properties.

In this setting, it is important to notice that a properly achieved property effectively becomes an assurance for the desired system behavior. Even though the MAPE-K loop—and its further refinements—can be considered as an important contribution for SAS engineering, we strongly believe that the lessons learned by control theory in the assurance of desired properties is the direction we need to follow in order to consolidate it as a milestone. The remainder of this section makes the case for this claim by showing how control considerations provide a basis for reasoning about the control properties of systems.

6.3 Properties of the Open Loop Model

One of the most important reasons for having a model is to study the properties of the system it models. In the open loop model, the most important properties are stability, observability and controllability. Concerning assurance, we have to upfront understand their impact and implications to design the system such that required assurance can be obtained at all.

6.3.1 Stability

In simple terms, stability means that for bounded inputs (commands or perturbations), the system will produce bounded state and output values. In the case of Example 1 (cf. Sect. 5.1.1), a stable system might mean that for keep-alive connection intervals of 10 min (u_2) and for a 100 connections (u_1), the response time of the server is guaranteed to be between 1 and 2 s. Formally, stability is proven by necessary and sufficient conditions and, given a model, one can use Bode and Nyquist plots [4] to study stability. Examples of stability studies in control of software and computing systems have been presented in [3, 29]. Unfortunately, perturbations are the enemy of stability in open loop SAS, because the system is not set up to recognize how much the perturbations affect the system. If the open SAS is not stable, it can be stabilized through the design of a suitable controller. However, by analyzing the stability of the open system, we understand the source of instability and design the controller appropriately. For example, it is known that a source of instability is a saturated queue. It can lead to performance instability, faults and crashes. For a complex SAS, we have to assume which queue has the potential to be saturated by external disturbances, and then we design a controller that computes u such that avoids saturation.

For discrete open loop models, in contrast to continuous open loop models, often no metrics for the output y and external disturbance p exists and thus the concept of stability is not applicable.

6.3.2 Observability

Observability is the property of the model that allows to find, or at least estimate, the internal state variables of a system from the output variables. This property is important from a pragmatic point of view. In a real system, it is impossible, hard or impractical to measure all state variables. On the other hand, the commands and outputs are easier to measure. By knowing the model of the system in the form of Eqs. (3) and (4), if the matrices A and C are well behaved, one can compute x as a function of y. Formally, a model is observable if the observability matrix

$$O = [C\, CA\, CA^2, ..., CA^{n-1}] \tag{12}$$

has the rank n, where n is the number of the state variables in x. Matrix O has to be invertible (or have rank n) in order to compute x as a function of y. Its structure is determined by writing the Eq. (4) for n values of k and considering D = 0. Simple algebraic operations will allow us to compute x only if the matrix O is invertible. Examples of web service observability from a performance point of view can be found in [15]. Examples of how to estimate software performance parameters for applications deployed across multi-tiers and using Kalman filters are presented in [65].

The concept of observability can be transfered to discrete open loop models by considering whether the observable output y provides enough information to determine the state x. However, unless the start state is not known when the controler starts, the typical criteria employed is controllability in the sense that the existence of a controller to achieve the control objective is directly considered.

6.3.3 Controllability (or Reachability)

The concept of controllability (or state controllability) describes the possibility of driving the open system to a desired state, that is, to bring its internal state variables to certain values [15]. Of course, the system can be driven to a certain state by changing the command variables u. In the case of Example 1 (cf. Sect. 5.1.1), we should be able to find the values of u_1 and u_2 that will bring the server utilization (x_1) and memory usage (x_2) to 50%, for example. Like observability, the formal proof is given by the structure of the matrices that make the Eqs. (3) and (4): the system is controllable if the controllability matrix,

$$S = [B\; AB\; A^2B, ..., A^{n-1}B] \tag{13}$$

has the rank n, where n is the number of state variables. The structure of matrix S is determined by writing the Eq. (4) for n values of k. Simple algebraic operations will lead us to matrix S and we can compute x as function of u only if S is invertible, hence has the rank n. If observability is not a necessary condition for designing a controller for SAS, the Controllability property is. Even we do not have a explicit open loop model, a qualitative analysis should be done. If the external disturbances, their frequency and amplitudes are known, do we have enough control inputs to compensate those perturbations?

Actually, the typical criteria employed for discrete control is controllability in the sense that the existence of a controller to achieve the control objective is directly considered. However, this perspective often differs from controllability in the above introduced sense. E.g., the earlier outline objective to stay within the set of safe states considered for controller synthesis does not equal controllability, which would only demands that the control input allows that any state can be reached, but not that the open loop model can then be forced to stay with a particular set of states.

6.4 Complex Open SAS and Model Composition

It is always the case that software systems are made of many components, interconnected together. The components might have different life cycles and be under different administrative domains. It is therefore imperative to answer the question: What are the properties of the overall open system (controllability, observability, stability) if we know the properties of the individual components? If the components are described by models such as the ones presented in Eqs. (3) and (4), then we have a method to answer that question. Models such as (3) and (4) can be composed, that is, if they are connected in series, parallel or through a feedback loop, a resultant composed model can be derived from the individual models. The composed model will have the same structure as the one presented in Eqs. (3) and (4), but the matrices A, B, C, D will be different but determined analytically from the individual models. Two services connected in series means that the output of the first one is the input of the second one (inputs and outputs have the control theoretic meaning). Parallel composition means that the same input reaches two different models and the output of the models will be summed up and used (eventually) as an input in other model. Feedback composition refers to a composition in which the output of a model is brought back and subtracted from the input of another model, as illustrated in Eqs. (7) and (8).

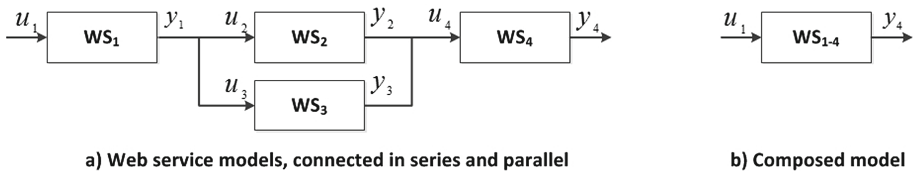

Fig. 11. Model composition for web services.

Example 3: Web service composition. In [52], the authors showed, based on web service performance case studies, how web services models can be composed in series and in parallel. Figure 11 illustrates the composition process: a set of web services are interconnected in an workflow; if we consider their performance, then the inputs (commands), u, are arrival rates and the outputs, y, are throughputs. For services in series, the throughput of the service on the left becomes the arrival rate for the service on the right. In Fig. 11a, WS_2 and WS_3 are connected in parallel, which means $u_2 = u_3 = y_1/2$ and $u_4 = y_2 + y_3$. By composing the models using their property of linearity, we can get a model of the entire system that is similar to the one presented in Eqs. (3) and (4). It is interesting to note that by composing web service models, the properties of the individual models (stability, controllability, observability) are not necessarily transferred to the composed model [15]. Model composition through feedback loops was briefly described by Eqs. (7) and (8).

6.4.1 Discrete Control

Also in case of discrete finite state models hold that composition would be necessary if larger systems should be considered. Furthermore, also in this case the composition usually does not preserve often desirable properties. Even worse oftentimes the composition can itself already lead to unwanted phenomena such as deadlocks. It is also interesting to note that for models that result from the composition of multiple transition systems oftentimes the time until which an control input may have impact on the state or output in the envisioned manner may become quite large, as each internal interaction between the composed transition systems leads to delays that sum up. In addition, the composition leads quite fast to very large state spaces as the orthogonal combination of the states of the composed transition systems has to be considered.

6.5 Properties of the Closed Loop Model

Once an open SAS has been analyzed and the designer has a good understanding of its stability, observability and controllability, a controller can be designed. When an explicit model of the open loop is available, the closed loop model can be synthesized mathematically to achieve the properties the designer wants. As Eqs. 7 and 8, suggest, A_c can be shaped and therefore the properties of the closed loop model. In general, the controller is designed to achieve some goals. The most frequent ones are described below. If assurance needs fit to one of these types of goals, control concepts can be employed to obtain the required assurance.

6.5.1 Stability

It refers to whether control corrections move the open system state, over time, toward the reference value. A system is unstable if the control causes overcorrections that never decrease, or that increase without limit. Instability can be introduced by making corrections that are too large in an attempt to achieve the reference level quickly. This leads to oscillating behaviors in which the system overshoots the reference value alternately to the high side and the low side. The opposite problem from stability is sluggish performance: if the control corrections are too small, it may take a very long time for the system to reach the reference value. In designing a control algorithm, the goal is to make the largest control correction that does not make the system unstable. Of course, the more we know about the way the system responds to changes in the tuning parameters (*i.e.*, the dynamics of the system), the easier it is to accomplish this goal.

If a model of the open loop exists, Eqs. (3) and (4), a controller can be designed to correct any instability of the open system that might be caused by perturbations or intrinsic properties of the software. The stability property is defined for the entire composed system, (considering y_r or p as input and y as output). With a model available, the stability is prescribed in the frequency domain (Z or S transforms) and implies solving a set of algebraic equations that will give the desired controller equations. This has been recently shown for several SAS use cases. For example, a controller design for stability is presented by Cortellesa *et al.* [3] for a performance case study.

As in the case of the open loop model, for discrete closed loop models, in contrast to continuous closed loop models, often no metrics for the output y and external disturbance p exist, and thus the concept of stability is inapplicable.

6.5.2 Robustness or Robust Stability

A special type of stability in control theory is the *robust stability*. Robust stability means that the closed loop system is stable in the presence of external disturbances, model parameters and model structure uncertainties. If we refer to models (3) and (4), external disturbances uncertainties refer to wide variations of p in amplitude and frequency. Consider an application in which the disturbance is the external load, that is the number of users and the frequency of user requests. The number of users can vary from 100 to 1 million and their requests frequency from 0.1 to 40 requests per second. Can we design one single controller that maintain the performance of the system (response time) below a certain threshold in the presence of those large variations? Model parameter uncertainties in Eqs. (3) and (4) refer to errors in identifying A, B, C, D, F. How can we design a controller that can achieve stability when there are large errors in those parameters? Model structure uncertainties refer to working with models (3) and (4) that miss dynamics of the system (for example ignoring important state variables). Can the controller provide stability in this situation? In general, can we design one static controller that assure stability within quantifiable uncertainties bounds?

In control theory, there are several design techniques that help a designer achieve robust stability (*e.g.*, H_∞, loop shaping or quantitative feedback theory or QFT). When a SAS can be described by a system of Eqs. (1) and (2) or (3) and (4), then we should consider those control techniques. For more complex SAS, it is probably feasible to consider Adaptive and/or Hierarchical Control where multiple controllers are synthesized or tuned dynamically to face wide disturbances.

For discrete models, in contrast to continuous linear models, often a single event that is different is sufficient to completely change the subsequent observable behavior. Consequently, robustness is in general difficult to achive for these kind of models. Therefore, even if metrics for the output y and external disturbance p exists, robust stability is often not reasonable to expect in case of discrete models.

6.5.3 Performance

In addition to stability, a controller has to achieve certain performance metrics: rise time, overshoot, settling time, steady error or accuracy [4,8], as illustrated in our web service for computing definite integrals. For a system subject to a PID controller, such as the one treated in that example, all of these metrics depend on the selected values for coefficients K_p, K_i, K_d. Figure 12 indicates these metrics,

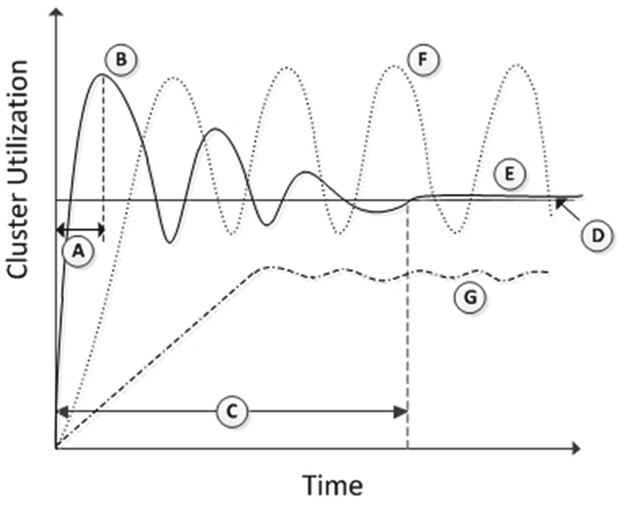

Fig. 12. Quality metrics for a controlled system [8]: rise time (A); overshoot (B); settling time (C); steady error or accuracy (D). Fine-tuned values for a PID controller, K_p, K_i, K_d, result in a smooth behavior reaching the reference level with accuracy and acceptable settling time (curve labeled (E)). Large values for K_p (and in lesser extents K_i and K_d), produce corresponding large oscillations around the reference level (curve labeled (F)). Small values usually result in a sluggish behavior (curve labeled (G)).

and illustrates the cluster utilization of a web service when perturbed by a workload change.

Assume the reference level is the horizontal continuous line in the figure. The perturbation, that affects the system at the origin of the time axis, drops the utilization to close to zero. A good controller (PID in this case) will compute a command u (the number of replicas in the cluster) that will bring the web service behavior back to normal in a short time (rise time) by varying the number of replicas in the cluster. Very likely, the system will be under- or over-provisioned, but the over- and undershooting must be kept small since it can affect other attributes, like response time and stability. Naturally, a few oscillations around the reference level are acceptable, but after a time (settling time), preferably short, the cluster utilization must reach a stable behavior (cf. curve labeled (E) in the figure). The figure also presents two examples of unstable and unacceptable control behavior: one which keeps oscillating around the reference level, and one which never reaches it (cf. curves labeled (F) and (G), respectively).

For discrete models, in contrast to continuous models, we often have no metrics for the output y, and thus, many of the above introduced terms cannot be employed. However, there are cases such as the synthesis of strategies for games, where the worst-case number of steps required to win the game could be seen as a criteria for performance.

6.5.4 Optimality or Linear Quadratic Regulator (LQR)

In many cases a controller has many goals, especially when there are multiple reference points, multiple input variables and multiple outputs. For example, a cruise controller should track the set speed but also minimize the consume of fuel. Usually the goals are conflicting. In control theory, the common way to treat this is to define an objective function across different goals. The goals can have different weights. Most commonly, the goals are defined across the state, output and control variables in the form of a quadratic function. The role of the controller is to find the commands u that minimize this quadratic function.

This type of controller with multiple goals is especially important for SAS systems, especially when the goal of the controller is a combination of conflicting goals, such as maintaining performance within certain bounds, optimizing the use of resources for saving cost and energy, and maximizing user experience. For example, Ghanbari *et al.* [28], for a SAS described by equations similar to (3) and (4), defined a quadratic function over the commands u (number of service instances) and the deviations from the setpoint y_r (or SLA violations) for cost and performance optimization in cloud computing, with Q and R matrices that penalize some elements of y and u:

$$J = \sum_{t=0}^{k} (y - y_r)^T Q (y - y_r) + u^T R_u \tag{14}$$

They showed that a controller can be designed to optimize the sum of two conflicting goals (performance and cost).

In control theory, there are algorithms that assure the optimality (or sub-optimality) of the solution and those can applied to SAS that can be described by Eqs. (3) and (4). However, the definition of the objective function, practical ways to establish the weights of the goals, are subject of further studies.

6.5.5 Look-Ahead Optimality or Model Predictive Control (MPC)

In control theory, MPC optimizes a set of goals over a future time horizon H. This makes sense for economic goals, where variables such as the revenue or profit are calculated over a time horizon. The idea behind these controllers is to make strategic decisions, that is to make a current decision by taking into account the long term goals. In other words, a decision may be suboptimal now but may prove optimal in long term. The goal of the controller in this case is to find $u(k), u(k+1), ..., u(k+H)$, with k the current time that optimizes a function similar to LQR over the future horizon H.

This type of optimization is especially important to complex SAS that have multiple design goals and have to prove efficiency over long periods of time. Examples of such optimizations, mostly in server provisioning for performance, cost and power can be found in [27,35,64]. The optimization problem for MPC problem looks like:

$$J = \sum_{t=k}^{k+H} (y - y_r)^T Q (y - y_r) + u^T R_u \tag{15}$$

In essence, a controller that optimizes J performs the following: if the current time is k, then the controller computes the future states x and the future outputs y for the next H time intervals. At the same time, the controller computes the controls u for these future H time intervals such that the entire sum in J equation is optimized. That is possible given the Eqs. (3) and (4). Pragmatically, the controller only implements the control u(k) from the entire set of controls. At time k + 1, the controller recomputes the commands for k + 1,. . ..k + H + 1 and, again, implements $u(k + 1)$.

Optimality in discrete models can be defined in contrast based on the current state. If accumulated over the sequence of states the software system is in, we get an overall reward, which can be used as an optimization criteria for the synthesis of controllers. However, in case of non-determinism in the model (unknown effects due to p), the optimization can only take, for example, the worst or best case into account. Therefore, oftentimes probabilistic models are considered to then optimize the expected reward.

6.5.6 Additional Properties only Related to Discrete Control

The corresponding properties that are best managed by discrete control techniques have to do with subsets of a state space, characterized by predicates, or paths in the state space described by allowed and forbidden sequences of transitions. Such properties can be checked on a given transition system, representing the behavior of a program or system, *e.g.*, using model checking algorithms [17] for which there exist numerous tools. Some can also be submitted to synthesis, automatically producing an implementation satisfying the property *e.g.*, [9]. Such properties can be specified using formulae in temporal logics (*e.g.*, LTL, Linear Time Logic). *Safety* properties concern characteristics that should always, or never, occur, and typically concern reachability (or not) of some state (or states set, defined by a predicate), as for example two tasks accessing the same resource. They can also concern path properties, involving sequences of observations or of actions, such as elevator doors opening should never occur between starting and stopping of the vertical movement. Such safety properties can be the object of synthesis in the framework of supervisory control for DES [14], in an approach where the synthesized controller constrains the values of controllable variables so that, for all sequences of uncontrollable inputs values, the paths will keep satisfying the property (for examples, see [48]). Another type of property, for more expressive logics, concerns *liveness*, where something can eventually become true.

6.6 Open Questions

Control theory models provide fundamental ideas that can be used to leverage assurances in SAS adaptation mechanisms. The open and closed loop properties, even if not captured in formalized models for SAS, must be considered by SAS designers along the system's engineering life cycle. Villegas *et al.* comprehensively define these properties in the context of SAS systems [58]. Implied questions are, for example:

– Above all, how can we determine whether a given SAS will be stable?
– How quickly will the system respond to a change in the reference value? Is this fast enough for the application? Control of a supersonic aircraft requires much faster response times than control of a slow-moving ground vehicle. Are there lags or delays that will affect the response time? If so, are they intrinsic to the system or can they be optimized?
– Are there some bounds of the external disturbances we know and how to design the SAS for?
– Can we design a robust controller to achieve robust stability or use Adaptive or Hierarchical Control?
– How accurately shall the system track the reference value? Is this good enough for the application?
– How much resources will the system spend in tracking the reference value? Can this amount be optimized? What is more important, minimizing the cost of resources or tracking the reference values? Can this tradeoff be quantified?
– It is very likely that multiple control inputs are needed to achieve robust stability. How will the control algorithm select the corrections to the control inputs?

7 Assurance Challenges for Self-Adaptive Software

From the examples presented in the previous sections, it should be clear that the concepts and principles of control theory and the assurances they provide, at least in abstract form, can be applied to a large class of problems in SAS. In the realm of SAS systems, we must consider at least two scenarios.

The first, in which it is possible to apply control theory *directly* to SAS, that is, by building a mathematical model (*e.g.*, a set of Eqs. (1) and (2) or automata) for the software system behavior, and applying control theory techniques to obtain properties, such as stability, performance, robustness (or robust stability), or safety and lifeness properties. These properties automatically grant corresponding assurances about the controlled system behavior. This scenario requires two main necessary conditions in case of continuous linear models: the problem must refer to quantitative metrics, and it must be simple enough to be modeled as a linear set of equations relating control inputs, outputs and state variables in the open and controlled system.

In the case of discrete models, the scenario requires also two conditions: the problem must be of a logical nature, involving conditions and synchronization, such that it can be formulated as a controller synthesis problem, and must be simple enough to be modeled as a not too-large transition system. Nonetheless, there are still no clear guidelines about the limitations of control theory as directly applied to SAS systems in the general case.

The second scenario arises when it is infeasible to build a reasonably precise mathematical model, but instead, it is possible to create an approximated operational or even qualitative model of the SAS behavior. In this case, the formal definitions and techniques of control theory may not apply directly, but understanding the principles of control theory can guide the sorts of questions the designer should answer and take care of. In other words, even if a mathematical model is unavailable, control theory provides validation obligations and elements that can be exploited to model the phenomena to be controlled.

In this section, we highlight the main challenges a designer faces when dealing with the problem of designing SAS systems, and extrapolate the questions and concepts from the control theory approach into the general case of SAS.

7.1 Modeling Challenges

In both of the aforementioned scenarios, and based on the specification of the system properties to achieve, the SAS system designer must tackle the following main modeling challenges:

- *How to implement and instrument the control commands u* in the software system to be controlled? For instance, Hellerstein *et al.* control the number of active threads in the Apache thread pool by varying the respective parameter [29]. Even though this can appear as natural, it is unclear whether the Apache server was intentionally designed with this parameter to be modified automatically or rather to be configured manually at installation time. For any other given software system to be controlled, what parameter should be defined, and affecting what components and functionalities? Is the provision of one (or several) parameters the best alternative for implementing the control commands? Other options, very different to varying numerical parameters, are modifying the software structure in critical parts (*e.g.*, using domain-specific design patterns), as in [55]; or using machine-learning strategies to modify the system behavior, as in [23].
- *How to obtain an operational understanding of the effects* that control commands will have on the controlled system? That is, a variation of one unit on the control parameter affects in what magnitude the number of related components and the behavior of their functionalities?
- Given a set of control commands and the operational understanding of their effects on the system behavior, how to refine these commands to refine also the granularity of their effects, in order to improve the accuracy controllability?
- How many cycles of control are needed to reach the reference level fast? How to determine the change size to be applied in each control cycle to minimize overshoot?

7.1.1 The Core Phenomena to Control

In control theory, differential (or difference) equations are the models on which principles and properties of the phenomena to control are described and analyzed.

Depending on the behavioral characteristics of the target system, a controller can be defined to make the system behave as desired. The desired behavior is usually specified by either providing a reference input the system should follow (*e.g.*, PID controllers), or as an optimization problem (*e.g.*, Model Predictive Control) [5,43]. The characteristics of a controller are usually defined by adjusting its parameters, which have special significance to generate the signals that will adapt the system depending on how far measured outputs are from the corresponding reference inputs.

Indeed, controllers are designed as parametric functions that generate the control signals that drive the target system towards accomplishing its goals. In software systems the identification of the core phenomena to control is typically a complex task. In contrast to physical systems, software systems still lack general methods to model the multi-dimensional and non-linear relationships between system goals and adaptation mechanisms [46,58]; moreover, the problem and solution spaces can differ drastically (*e.g.*, in the web service example for computing integrals of elementary functions, calculating $\int \frac{x}{\sqrt{x^4+10x^2-96x-71}}\,dx$ takes polynomial time, but only changing the constant 71 to 72, that is, $\int \frac{x}{\sqrt{x^4+10x^2-96x-72}}\,dx$, takes exponential time, as analyzed in [11]). This simple example illustrates how complex the problem space can be in SAS systems, on which tracked metrics depend upon, directly and explicitly.

Furthermore, an adaptation mechanism can reconfigure the software structure by applying domain-specific design patterns with the goal of improving the system performance, as realized in [55]. Examples of these design patterns and strategies include Leaders/Followers, Half Sync/Half Async, Load Balancer, and Master/Worker, among others [2,7,13,40]. However, it is still an open challenge to model the exact effect of the structural and behavioral elements introduced by a pattern in the system performance.

Assurance Challenges. In general, the analysis of the system model should determine whether the "knobs" have enough power (command authority) to actually drive the system in the required direction. Many other research questions remain open in the identification and modeling of the core phenomena to control in software systems. For example, *how to model explicitly and accurately the relationship among system goals, adaptation mechanisms, and the effects produced by controlled variables* when the control variables are as complex as a pattern and workflow? Can we design software systems having an explicit specification of what we want to assure with control-based approaches? Can we do it by focusing only on some aspects for which feedback control is more effective? Can we improve the use of control, or achieve control-based design, by connecting as directly as possible some real physics inside the software systems? How far can we go by modeling SAS systems mathematically? What are the limitations?

7.1.2 Sampling Period

One important decision when building a model is the sampling rate, that is, the frequency with which the states, outputs and commands are monitored and processed. In terms of our models (1) and (2), what is the difference between time k and $k + 1$? The meaning of "1" above was "the next sample", but after how many microseconds or minutes is the next sample? For models, such as (1) and (2), and for quantitative software qualities that can be approximated with continuous signals (performance, cost, energy), Nyquist-Shannon sampling theorem [50] can provide some practical guidelines. Simplifying, the theorem says that if the highest signal harmonic one wants to reconstruct from the sampling data has h Hertz, one should sample with a minimum frequency of $2h$ Hertz.

In Zheng *et al.* [66], the authors showed that for the specific performance model they were building, a sampling rate of five minutes was appropriate. Note that in [66] the authors were interested in mean response time and server provisioning decisions, which are rare control inputs. For higher frequency models and decisions (like changing the number or threads), that sampling rate is not enough. Further, Solomon *et al.* [53] showed how to find the sampling rate from the cutoff frequency. The cutoff frequency is the frequency of the command u that has no effect on the system (one can imagine that, for the example described by Eqs. (5) and (6), by adding and removing an HTTP connection at very high frequency, on average, has limited effect on the response time because the connection cannot be used). Solomon *et al.* use Bode plots to analyze models like (3) and (4) in the frequency domain and then determine the cutoff frequency. The sampling rate is then twice the cutoff frequency. It is obvious that the sampling rate depends on what we monitor and control, but it is unclear how to determine it in many cases and how to guarantee its correctness. Moreover, in SAS, most events are discrete and, thus, it makes more sense to use the number of events (such as number of failures or number of requests) instead of a sampling rate.

Assurance Challenges. The identification of the optimal sampling rate is a non-trivial task that can negatively impact the quality of the decisions taken by the controller. How can we ensure an optimal sampling rate over time? What is the overhead introduced by oversampling the underlying system? Can we control the sampling rate depending on the current state of the SAS?

7.2 Composition and Incrementality: V&V Tasks

Considering the scenarios we introduced for applying control theory to the engineering of SAS, validation and verification (V&V) is highly relevant, especially for the scenarios in which obtaining mathematical models is infeasible. If it is impossible to guarantee desirable properties based on the control theory principles and techniques, at least we should consider introducing V&V tasks on critical properties—an aspect that is commonly omitted in self-adaptive software proposals [58]. Nonetheless, performing V&V tasks (*e.g.*, model checking) over the entire system–at runtime, to guarantee desired properties and goals, is often infeasible due to prohibitive computational costs. Therefore, one fundamental requirement for the assurance of SAS systems is for these V&V tasks to be composable and applicable incrementally along the adaptation loop, among other conditions, as analyzed in [56].

Assurance Challenges. In this regard, relevant research questions include: *Which V&V tasks can guarantee which control properties*, if any, and to what extent? *Are stability, accuracy, settling-time, overshoot and other properties composable* (*e.g.*, when combining control strategies which independently guarantee them)? What are suitable techniques to realize the composition of V&V tasks? Which approaches can we borrow from testing? How can we reuse or adjust them for the assurance of SAS systems? Regarding incrementality: in which cases is it useful? How can incrementality be realized? How do we characterize increments, and their relationship to system changes?

7.3 Timing Issues and Lags

As discussed previously, a major challenge in designing a control algorithm is to determine what is the largest control correction that does not destabilize the system. In addition to stability, SAS is affected by specific characteristics of the system dynamics, such as lags (when the system responds only slowly to a change in a tuning parameter), delays (when it takes time before a change takes any effect, such as startup time when adding a server), and differences between transient response (when response to a change in environment decays over time) and steady-state response (when the change is enduring). If the open and closed SAS are described by equations such as (3), the time lag should be included in the equations. The main problem in SAS is that lags are highly variable; it is hard, if not impossible, to bound them.

Another difference is that, in contrast to physical plants, behavior load in software systems sometimes presents no inertial effect. For example, in the room temperature control case, the trend of the difference between the reference level y and the measured output y_r usually augments or diminishes quite slowly. However, in software systems, all of the service requests can disappear instantly, for instance if all users cancel their requests, or the Internet connection of the web server is interrupted. Besides, while most physical processes do not respond immediately to changes in the control parameters, most software systems do not respond so precisely to control changes.

Even in the absence of a mathematical model, it is clear that the designer needs a good understanding of at least the direction of change that will result from a given control input, and preferably a sense of its magnitude. Since SAS systems often have more than one control input, the absence of a mathematical model makes understanding the relations among the inputs, disturbances, and lags more difficult, but no less important.

Lags, such as those that arise from bringing another server online, introduce some degree of uncertainty in SAS systems. Under such conditions, how can we assure the appropriate synchronization of control actions and SAS reactions? Can we guarantee the timing required by the software system to operate the change ordered by the control algorithm?

7.4 Challenges in Control Strategies Design

It is clear from the above discussion that decisions about the control strategy are absolutely critical to the design of SAS. Classical control theory operates in settings that are simple enough that mathematical analysis is often possible. SAS, on the other hand, are usually too complex to characterize completely, are highly non-linear and time-variant. This makes the design of the control strategy even more critical.

We often describe control in terms of strategic changes to the reference value or the level of external disturbances. It is almost impossible in SAS to design a controller that can work well with all possible values of references or disturbances. This is another reason to make explicit design decisions about the control strategy. When the system is time variant (that is, the matrices A, B, C, D, F in Eqs. (3) and (4) are time dependent) or the reference and disturbances change over large ranges, the design goals are achieved by engineering an adaptive controller that is tuned on-line (adaptive control), based on the operational point of the system. Adaptive control can be achieved using various strategies, such as Model Identification Adaptive Control (MIAC) or Model Reference Adaptive Control (MRAC) [38]. Those strategies are of paramount importance to SAS systems design.

7.4.1 Modeling External Disturbances and Uncertainties

In classical control theory, the feedback loop is designed with knowledge of the nature of the external disturbances, their characteristics and bounds. Furthermore, the design of the feedback system relies on sensed values about internal state or system behavior. This design must consider explicitly how accurately those sensed values actually represent the true state of the system. In the cruise control example, the control engineer knows that the speed of the car (the controlled output) is altered by friction, road conditions, hills and valleys, and all those can be captured in a model (like the term $F * p(k)$ in Eqs. (3) and (4)). Once a set point is set (*e.g.*, 100 km/h) the controller has to respond adequately (*i.e.*, with certain accuracy and overshoot) to changes in perturbations. These changes refer to both amplitude (the slope of the hill) and frequency (alternations of hills and valleys). Similar considerations are taken into account for thermostat control or autopilots.

In SAS, external disturbances are harder to identify and model. In addition, in SAS external disturbances are most likely to change, not the reference values, so we have to design feedback loops that respond well to external disturbances. Consider an application deployed in a public cloud or an application using services provided by third parties. Assuming a feedback loop that maintains the response time at a setpoint, what are the external disturbances that affect the response time? We can make some assumptions about application load (number of users or their distributions and requests during a day), but what about the external disturbances affecting the cloud or third party services? Indirectly those disturbances are going to affect the application response time. How do we identify all those disturbances, their amplitude and frequency, how do we model or take their influence into account when we assure our feedback loop?

7.4.2 Complex Reference Values

It is tempting to set a reference level as a combination of goals. For example, to keep wait time below some threshold and energy consumption below some other threshold. Selecting a composite reference point exposes the risk of creating a situation in which one term of the reference value calls for increasing the tuning parameter and another term calls for decreasing the reference parameter. For example, if both wait time and energy consumption are limited, the former might lead to activating a server while the latter demands deactivating a server. In the case of the home thermostat, maintaining a comfortable temperature and saving energy consumption are clearly two goals that may lead to a conflictive situation. We must at minimum be aware of these conflicts; even better, we should refrain from creating them. In any event, thinking carefully about such conflicts can help avoid or manage the resulting complexity. One way to solve this is to use just one variable as a set point and use an optimization function like LQR to mediate among the other variables.

In the presence of complex reference values, can we detect such conflicting goals *a priori* or *a posteriori*? In case of conflicting goals, can we identify the constraints linking several goals in order to capture a more complex composite goal?

7.4.3 Management of Viability Zones

A viability zone of a SAS system can be defined as the set of possible states in which the system operation is not compromised with respect to a desired property [6]. Therefore, a SAS system may have more than one viability zone: at least one for the managed system and another for the controller or adaptation mechanism. The level of assurance of a SAS system may then be judged by taking into account the state of the system with respect to its viability zone(s). A viability zone can be characterized in terms of the properties to be satisfied, the quality attributes that describe them and corresponding desired values [59]. Moreover, viability zones can change according to variations in relevant context. Feedback control may also help in the specification of viability zones for SAS systems. Properties to be satisfied define the dimensions of a viability zone. These properties are usually characterized in terms of quality attributes that correspond to reference inputs (y_r) defining the boundaries of viability zones. The goal of adaptation mechanisms (commands u) is to maintain the managed system within its viability zone(s). Viability zones may be also dynamic, not only because relevant context and desired properties may change over time due to uncertainty, but also when the adaptation goal is optimization.

The definition of viability zones is not a trivial task, particularly because desired properties are usually characterized in terms of several variables (*i.e.*, quality attributes). Relevant research questions in this regard include: how many viability zones are required for the assurance of a particular SAS system? Does each desired property require an independent viability zone? How to manage trade-offs and possible conflicts among several viability zones?

The dynamic nature of viability zones is also a relevant research challenge in the assurance of SAS systems. Particularly, because it implies adjusting the domain coverage not only of design and adaptation realization but also of V&V tasks. Open research questions in this aspect include: what are runtime models that can be used for the incremental and dynamic derivation of software artifacts for implementing V&V tasks? How to maintain the causal connection between viability zones, adapted system, and its corresponding V&V software artifacts? How to adapt these artifacts at runtime?

8 Conclusions

In this chapter, we explored control theory applications to self-adaptive software (SAS) design. We started with an overview of the SAS and control theory concepts, summarized the main research results of control theory applied to software and computing in general and then focused on control assurances. We described the properties of open loop models, the control strategies and goals, and then elaborated on assurance challenges.

There are several conclusions we can draw from this exploration. Control theory can be applied as is to a subset of SAS and for a subset of quality attributes, including performance, cost, reliability, energy consumption. Research results suggest that it is feasible to design a controller for relatively simple use cases, especially for single input/single output systems and for time invariant cases. Even for these metrics the research is still open, and more complex case studies are needed to draw general conclusions. To apply control theory, a specific type of model is needed. If a linear or discrete model can capture the outputs, control inputs, states, and eventually external perturbations of the open loop SAS, a controller can be synthesized.

Nonetheless, many complex SAS are time-variant and multi-dimensional, with multiple input and output control variables. For these cases, more advanced control theory concepts, such as Model Identification Adaptive Control (MIAC), are needed, where the model is identified at runtime and a controller is synthesized periodically. In even more cases, an explicit model for SAS cannot be constructed. Nevertheless, control theory general principles are still a good guideline to follow. The designer needs to understand the properties of open systems: inputs, outputs, perturbations, states, and how these influence each other. Properties, such as stability and controllability, should be investigated and understood before starting to design a controller (or adaptive/autonomic manager). The controller should be designed with some goals in mind: optimization, stability, performance and accuracy, and then tested and verified. A final conclusion is that the field of control theory applied to SAS and the related assurance challenges are still open, and constitute important research areas in the engineering of these kinds of software systems.

References

1. Abdelzaher, T., Stankovic, J., Lu, C., Zhang, R., Lu, Y.: Feedback performance control in software services. IEEE Control Syst. **23**(3), 74–90 (2003)
2. AlBahnassi, W., Mudur, S.P., Goswami, D.: A design pattern for parallel programming of games. In: 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), pp. 1007–1014. IEEE (2012)
3. Arcelli, D., Cortellessa, V., Filieri, A., Leva, A.: Control theory for model-based performance-driven software adaptation. In: Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures, pp. 11–20. ACM (2015)
4. Åström, K.J., Murray, R.M.: Feedback Systems. An Introduction for Scientists and Engineers (2008)
5. Åström, K., Wittenmark, B.: Adaptive Control. Addison-Wesley series in Electrical Engineering: Control Engineering. Addison-Wesley, Reading (1995)
6. Aubin, J.P., Bayen, A.M., Saint-Pierre, P.: Viability Theory: New Firections. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-16684-6
7. Babaoglu, O., Canright, G., Deutsch, A., Caro, G.A.D., Ducatelle, F., Gambardella, L.M., Ganguly, N., Jelasity, M., Montemanni, R., Montresor, A., et al.: Design patterns from biology for distributed computing. ACM Trans. Auton. Adapt. Syst. (TAAS) **1**(1), 26–66 (2006)
8. Balzer, B., Litoiu, M., Müller, H., Smith, D., Storey, M.A., Tilley, S., Wong, K.: 4th International Workshop on Adoption-Centric Software Engineering. In: Proceedings of the 26th International Conference on Software Engineering, ICSE 2004, pp. 748–749. IEEE Computer Society, Washington, DC (2004)
9. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Saar, Y.: Synthesis of reactive (1) designs. J. Comput. Syst. Sci. **78**(3), 911–938 (2012)
10. Braberman, V., D'Ippolito, N., Kramer, J., Sykes, D., Uchitel, S.: MORPH: a reference architecture for configuration and behaviour self-adaptation. In: Proceedings of the 1st International Workshop on Control Theory for Software Engineering, CTSE 2015, pp. 9–16. ACM, New York (2015)
11. Bronstein, M.: Integration of elementary functions. J. Symbolic Comput. **9**(2), 117–173 (1990)
12. Brun, Y., et al.: Engineering self-adaptive systems through feedback loops. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 48–70. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_3
13. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-oriented Software Architecture: A System of Patterns. Wiley, New York (1996)
14. Cassandras, C., Lafortune, S.: Introduction to Discrete Event Systems. Springer, New York (2008). https://doi.org/10.1007/978-0-387-68612-7
15. Checiu, L., Solomon, B., Ionescu, D., Litoiu, M., Iszlai, G.: Observability and controllability of autonomic computing systems for composed web services. In: Proceedings of the 2011 6th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI), pp. 269–274. IEEE (2011)
16. Cheng, B.H.C., et al.: Software engineering for self-adaptive systems: a research roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02161-9_1

17. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
18. Dahm, W.: Technology Horizons a Vision for Air Force Science & Technology During 2010–2030. Tech. rep., U.S. Air Force (2010), http://www.af.mil/information/technologyhorizons.asp
19. Delaval, G., Gueye, S.M.K., Rutten, E., De Palma, N.: Modular coordination of multiple autonomic managers. In: Proceedings of the 17th International ACM Sigsoft Symposium on Component-Based Software Engineering, CBSE 2014, pp. 3–12. ACM, New York (2014)
20. Diao, Y., Gandhi, N., Hellerstein, J.L., Parekh, S., Tilbury, D.M.: Using MIMO feedback control to enforce policies for interrelated metrics with application to the apache web server. In: Network Operations and Management Symposium, NOMS 2002. IEEE/IFIP, pp. 219–234. IEEE (2002)
21. D'Ippolito, N., Braberman, V., Kramer, J., Magee, J., Sykes, D., Uchitel, S.: Hope for the best, prepare for the worst: multi-tier control for adaptive systems. In: Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pp. 688–699. ACM, New York (2014)
22. D'Ippolito, N., Braberman, V., Piterman, N., Uchitel, S.: Synthesizing nonanomalous event-based controllers for liveness goals. ACM Trans. Software Eng. Methodol. **22**(1), 9:1–9:36 (2013)
23. Elkhodary, A., Esfahani, N., Malek, S.: FUSION: a framework for engineering self-tuning self-adaptive software systems. In: Proceedings of 18th ACM International Symposium on Foundations of Software Engineering, FSE 2010, pp. 7–16. ACM (2010)
24. Filieri, A., Hoffmann, H., Maggio, M.: Automated design of self-adaptive software with control-theoretical formal guarantees. In: Proceedings of the 36th International Conference on Software Engineering, pp. 299–310. ACM (2014)
25. Filieri, A., Maggio, M., Angelopoulos, K., D'Ippolito, N., Gerostathopoulos, I., Hempel, A.B., Hoffmann, H., Jamshidi, P., Kalyvianaki, E., Klein, C., Krikava, F., Misailovic, S., Papadopoulos, A.V., Ray, S., Sharifloo, A.M., Shevtsov, S., Ujma, M., Vogel, T.: Software engineering meets control theory. In: Proceedings of 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2015), pp. 71–82. IEEE Press (2015)
26. Gergin, I., Simmons, B., Litoiu, M.: A decentralized autonomic architecture for performance control in the cloud. In: 2014 IEEE International Conference on Cloud Engineering (IC2E), pp. 574–579. IEEE (2014)
27. Ghanbari, H., Litoiu, M., Pawluk, P., Barna, C.: Replica placement in cloud through simple stochastic model predictive control. In: Proceedings of the 2014 IEEE 7th International Conference on Cloud Computing (CLOUD), pp. 80–87. IEEE (2014)
28. Ghanbari, H., Simmons, B., Litoiu, M., Iszlai, G.: Feedback-based optimization of a private cloud. Future Gener. Comput. Syst. **28**(1), 104–111 (2012)
29. Hellerstein, J.L., Diao, Y., Parekh, S., Tilbury, D.M.: Feedback Control of Computing Systems. Wiley, Chichester (2004)
30. IBM Corporation: An Architectural Blueprint for Autonomic Computing, Technical report, IBM Corporation (2006)
31. Janert, P.K.: Feedback Control for Computer Systems. O'Reilly Media Inc., Sebastopol (2013)

32. Kalyvianaki, E., Charalambous, T., Hand, S.: Self-adaptive and self-configured CPU resource provisioning for virtualized servers using kalman filters. In: Proceedings of the 6th International Conference on Autonomic Computing, pp. 117–126. ACM (2009)
33. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Comput. **36**(1), 41–50 (2003)
34. Kramer, J., Magee, J.: Self-Managed systems: an architectural challenge. In: FOSE 2007: 2007 Future of Software Engineering, pp. 259–268. IEEE Computer Society, Washington, DC (2007)
35. Kusic, D., Kephart, J.O., Hanson, J.E., Kandasamy, N., Jiang, G.: Power and performance management of virtualized computing environments via lookahead control. Cluster Comput. **12**(1), 1–15 (2009)
36. Laddaga, R.: Guest Editor's introduction: creating robust software through self-adaptation. IEEE Intell. Syst. **14**(3), 26–29 (1999)
37. Laddaga, R.: Active software. In: Robertson, P., Shrobe, H., Laddaga, R. (eds.) IWSAS 2000. LNCS, vol. 1936, pp. 11–26. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44584-6_2
38. de Lemos, R., et al.: Software engineering for self-adaptive systems: a second research roadmap. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 1–32. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_1
39. Lennart, L.: Perspectives on system identification. Annu. Rev. Control **34**(1), 1–12 (2010)
40. Mattson, T., Sanders, B., Massingill, B.: Patterns for Parallel Programming, 1st edn. Addison-Wesley Professional, Reading (2004)
41. Müller, H., Pezzè, M., Shaw, M.: Visibility of control in adaptive systems. In: Proceedings of the 2nd International Workshop on Ultra-Large-Scale Software-Intensive Systems, pp. 23–26. ACM (2008)
42. Müller, H., Villegas, N.: Runtime evolution of highly dynamic software. In: Mens, T., Serebrenik, A., Cleve, A. (eds.) Evolving Software Systems. LNCS, pp. 229–264. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-45398-4_8
43. Murray, R.M.: Control in an Information Rich World: Report of the Panel on Future Directions in Control, Dynamics, and Systems. SIAM, Philadelphia (2003)
44. Northrop, L., Feiler, P., Gabriel, R., Goodenough, J., Longstaff, T., Kazman, R., Klein, M., Schmidt, D., Sullivan, K., Wallnau, K.: Ultra-Large-Scale Systems-The Software Challenge of the Future. Technical report, Carnegie Mellon University Software Engineering Institute (2006), http://www.sei.cmu.edu/uls
45. Oreizy, P., Medvidovic, N., Taylor, R.N.: Runtime software adaptation: framework, approaches, and styles. In: Proceedings of the 30th International Conference on Software Engineering (ICSE 2008), pp. 899–910 (2008)
46. Patikirikorala, T., Colman, A., Han, J., Wang, L.: A systematic survey on the design of self-adaptive software systems using control engineering approaches. In: Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2012), pp. 33–42. IEEE Press (2012)
47. Ramadge, P.J., Wonham, W.M.: Supervisory control of a class of discrete event processes. SIAM J. Control Optim. **25**(1), 206–230 (1987)
48. Rutten, E., Marchand, N., Simon, D.: Feedback control as MAPE-K loop in autonomic computing. In: de Lemos, R., Garlan, D., Ghezzi, C., Giese, H. (eds.) Self-Adaptive Systems III. LNCS, vol. 9640, pp. 349–373. Springer, Cham (2016)
49. Salehie, M., Tahvildari, L.: Self-adaptive software: landscape and research challenges. ACM Trans. Auton. Adapt. Syst. **4**, 14:1–14:42 (2009)

50. Shannon, C.: Communication in the presence of noise. Proc. IEEE **86**(2), 447–457 (1998)
51. Cowpertwait, Paul S.P., Metcalfe, Andrew V.: System Identification. In: Introductory Time Series with R. UR, pp. 201–209. Springer, New York (2009). https://doi.org/10.1007/978-0-387-88698-5_10
52. Solomon, B., Ionescu, D., Litoiu, M., Iszlai, G.: Autonomic computing control of composed web services. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2010), pp. 94–103. ACM (2010)
53. Solomon, B., Ionescu, D., Litoiu, M., Iszlai, G., Prostean, O.: Measurements and identification of autonomic computing processes. In: Proceedings of the 2010 IEEE International Conference on Computational Intelligence for Measurement Systems and Applications (CIMSA), pp. 72–77. IEEE (2010)
54. Sykes, D., Corapi, D., Magee, J., Kramer, J., Russo, A., Inoue, K.: Learning revised models for planning in adaptive systems. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE 2013, pp. 63–71. IEEE Press, Piscataway (2013)
55. Tamura, G., Casallas, R., Cleve, A., Duchien, L.: QoS contract preservation through dynamic reconfiguration: a formal semantics approach. Sci. Comput. Program. (SCP) **94**(3), 307–332 (2014)
56. Tamura, G., et al.: Towards practical runtime verification and validation of self-adaptive software systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 108–132. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_5
57. Truex, D.P., Baskerville, R., Klein, H.: Growing systems in emergent organizations. Commun. ACM **42**(8), 117–123 (1999)
58. Villegas, N., Müller, H., Tamura, G., Duchien, L., Casallas, R.: A framework for evaluating quality-driven self-adaptive software systems. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2011), pp. 80–89. ACM (2011)
59. Villegas, N.M.: Context Management and Self-Adaptivity for Situation-Aware Smart Software Systems. Ph.D. thesis, University of Victoria (2013)
60. Villegas, N.M., Tamura, G., Müller, H.A., Duchien, L., Casallas, R.: DYNAMICO: a reference model for governing control objectives and context relevance in self-adaptive software systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 265–293. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_11
61. Vromant, P., Weyns, D., Malek, S., Andersson, J.: On interacting control loops in self-adaptive systems. In: Proceedings of 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2011), pp. 202–207. ACM (2011)
62. Wang, Y., Lafortune, S., Kelly, T., Kudlur, M., Mahlke, S.: The Theory of Deadlock Avoidance Via Discrete Control. Principles of Programming Languages. POPL, Savannah, USA, pp. 252–263 (2009)
63. Weyns, D., et al.: On patterns for decentralized control in self-adaptive systems. In: de Lemos, R., Giese, H., Müller, H.A., Shaw, M. (eds.) Software Engineering for Self-Adaptive Systems II. LNCS, vol. 7475, pp. 76–107. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-35813-5_4

64. Zhang, Q., Zhu, Q., Zhani, M.F., Boutaba, R., Hellerstein, J.L.: Dynamic service placement in geographically distributed clouds. IEEE J. Sel. Areas Commun. **31**(12), 762–772 (2013)
65. Zheng, T., Woodside, M., Litoiu, M.: Performance model estimation and tracking using optimal filters. IEEE Trans. Software Eng. **34**(3), 391–406 (2008)
66. Zheng, T., Yang, J., Woodside, M., Litoiu, M., Iszlai, G.: Tracking time-varying parameters in software systems with extended Kalman filters. In: Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative research (CASCON), pp. 334–345. IBM Press (2005)