Automated Design of Self-Adaptive Software with Control-Theoretical Formal Guarantees

Antonio Filieri
University of Stuttgart
Stuttgart, Germany

Henry Hoffmann
University of Chicago
Chicago, USA

Martina Maggio
Lund University
Lund, Sweden

ABSTRACT

Self-adaptation enables software to execute successfully in dynamic, unpredictable, and uncertain environments.

Control theory provides a broad set of mathematically grounded techniques for adapting the behavior of dynamic systems. While it has been applied to specific software control problems, it has proved difficult to define methodologies allowing non-experts to systematically apply control techniques to create adaptive software. These difficulties arise because computer systems are usually non-linear, with varying workloads and heterogeneous components, making it difficult to model software as a dynamic system; i.e., by means of differential or difference equations.

This paper proposes a broad scope methodology for automatically constructing both an approximate dynamic model of a software system and a suitable controller for managing its non-functional requirements. Despite its generality, this methodology provides formal guarantees concerning the system's dynamic behavior by keeping its model continuously updated to compensate for changes in the execution environment and effects of the initial approximation.

We apply the methodology to three case studies, demonstrating its generality by tackling different domains (and different non-functional requirements) with the same approach. Being broadly applicable and fully automated, this methodology may allow the adoption of control theoretical solutions (and their formal properties) for a wide range of software adaptation problems.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Design—*Methodologies*; I.6.5 [**Computing Methodologies**]: Model Development—*Modeling methodologies*

General Terms

Design, Experimentation, Theory, Performance, Reliability

Keywords

Adaptive software, control theory, dynamic systems, non-functional requirements, run-time verification.

ICSE '14, May 31 - June 7, 2014, Hyderabad, India

1. INTRODUCTION

The growing complexity of computing systems is placing increased burden on application developers. This situation is made worse by the dynamic nature of modern systems, which can experience sudden and unpredictable changes; e.g., application workload fluctuations and system component failure. It is increasingly up to software engineers to manage this complexity and ensure applications operate successfully in dynamic environments.

The use of *autonomic* or *self-adaptive* techniques has been proposed to help engineers manage this burden. Such systems modify their own behavior to maintain goals in response to unpredicted changes. While adaptation of an application's *functional* aspects (i.e., semantic correctness) often requires human intervention, its *non-functional* aspects (such as reliability, performance, energy consumption, and cost) represent an important and challenging opportunity for applying self-adaptive techniques. For example, customers require continuous assurance of agreed performance and quality levels. These non-functional aspects can be managed by mapping them into specific *quantitative properties*. These properties can be measured and used to trigger adaptations guaranteeing requirements are met even in the face of unforeseen environmental fluctuations [11].

Such measurement-driven adaptation has been studied for decades in the context of *control theory*. Control systems have achieved widespread usage in many engineering domains which interact with the physical world. In such systems, the controller measures quantitative feedback from a *sensor* (e.g., a speedometer), and determines how to tune an *actuator* (e.g., a fuel intake) to effect the controlled *plant* (e.g., an engine). One major advantage of using control theory is that such techniques emit analytical guarantees of the system's dynamic behavior. In principle, adaptable software can be considered a controllable plant allowing control theory to be applied to self-adaptive software systems.

While researchers have applied notions from control theory to software systems, the control of software can still be considered in its very preliminary stage. There are many challenges that must be overcome to advance the application of control theory to software systems and many of these challenges arise from the difficulty of modeling the controlled systems [16, 17, 72]. Specifically, software applications have complex, often non-linear, interactions with the hardware and system software that support their execution. In addition, dynamic changes, due to maintenance or workload fluctuations, may invalidate a previously effective model.

This difficulty in defining concise and precise models of software behavior usually leads to the design of controllers focused on particular operating regions or conditions, or ad hoc solutions which address specific computing problems using control theory, but do not generalize [51, 64, 65]. For example, Hellerstein et al. define a controller for .NET thread pools that is not straightforwardly adapted to other architectures, though the high level task is quite similar [29]. Furthermore, this lack of generality requires application developers to spend significant development time designing and implementing control systems. Thus, there is a need for generalized methods which can automatically synthesize control systems that compensate for shortcomings in system models due to non-linear component interaction or dynamic fluctuations in the environment.

We address this need by presenting a methodology which automatically builds suitable system models and then uses those models to synthesize a controller suitable for the self-adaptive management of non-functional application requirements. Given a software system and a non-functional requirement (e.g., performance, accuracy, energy), our methodology first uses a training phase to generate a linear model of the system and then synthesizes a configurable controller. The controller overcomes potential non-linearities using a Kalman filter to adapt the linear model dynamically. In addition, for drastic changes in system behavior, the controller incorporates a change-point detection strategy to trigger an online model rebuilding phase. This methodology is general in the sense that it allows users to apply control theoretic techniques to a variety of scenarios without requiring the users to be control experts. Thus, general users can benefit from the formal guarantees of control systems without being experts themselves. Critically, the methodology synthesizes these controllers without a priori knowledge of system.

We evaluate our methodology in two ways. First, we perform a formal assessment of the guarantees it provides. Second, we perform an empirical assessment of the methodology on three different software applications: video compression, energy efficient resource provisioning, and dynamic binding and delegation. Similar problems appear as case studies in the literature concerned with self-adaptive software [61], and are here dealt with using our generalized approach.

2. CONTROLLING SOFTWARE

Our goal is to ease the development of self-adaptive software systems by automatically synthesizing a control system capable of managing non-functional aspects of the software's behavior. Toward that end, this section presents background on essential properties of self-adaptive software and relates them to analogous concepts in control theory.

We begin with an existing software system and some non-functional aspect which we want the software to self-manage. We assume no knowledge of the internals of the software system. Instead, we assume that 1) the system allows quantitative measurement of the specified non-functional aspect and 2) there is some tunable parameter of the system that affects that aspect. Our methodology first derives a model mapping the parameter's settings into expected feedback. Then, the methodology synthesizes a control system that uses the derived model to ensure the specified aspect achieves the desired quality of service.

We therefore refer to the adaptable software as our *plant*. The combination of the original software and synthesized controller is called the *controlled* system. The controller continuously determines the value of a *control variable*, which represents a setting for the tunable parameter.

We assume that the software's user expresses a goal, or *setpoint*, representing the operating target for the specified non-functional aspect of the controlled system, e.g. failure probability, response time, energy consumption, or a convenient combination thereof. From the perspective of software engineering, a controller should be able to provide the following properties [23, 28]:

- *Setpoint Tracking.* The self-adaptive system should achieve the user-specified setpoint. Furthermore, if a user changes the setpoint, the controller should drive the system toward a new setting satisfying the new requirement. For example, consider a system that self-manages the quality of a video streaming service. The system may have one setpoint for premium users, who receive high definition quality, and another for normal users, who are served with reasonable quality depending on available system resources. Setpoint tracking refers to the property that the self-adaptive system achieves the goal.
- *Disturbance rejection.* Disturbance rejection refers to the property that the self-adaptive system maintains the setpoint despite unpredictable deviations from expected behaviors; e.g., fluctuating load conditions or hardware failures. In addition, the system should not react to short-lived, transient external forces. For example, a controlled software system should not react to every cache miss, or the occasional page fault. Instead, a self-aware system should be able to distinguish between a condition of chronic page faults that effect performance and a single page fault that will not have a lasting effect on the system.
- *Robustness to inaccurate measurements.* Quantitative assessment of the running system usually relies on monitoring and/or other measurement procedures. Each of these might be subject to temporary biases, be affected by noise, or might require a certain time to converge to a convenient accuracy. A controller should provide a reasonable behavior even in presence of transitory errors on measured values. Besides reducing the sensitivity to measurement errors, robustness allows for the use of less invasive monitoring instruments, sometimes required for high accuracy but expensive in terms of performance overhead.

Not surprisingly, all of these properties have counterparts in control theory. In particular they can be mapped to (a combination of) the following four properties of the controlled system [28]:

- *Stability.* A control system is asymptotically stable if there exists an equilibrium point to which the system tends; i.e., for any given input, the output converges to a specific value (within a convenient accuracy). As time tends to infinity, the distance to the equilibrium point tends to zero. If the equilibrium point is determined by a setpoint, whenever the setpoint is reachable, an asymptotically stable system will converge to the setpoint while an unstable system would not.

- *Absence of overshooting.* An overshoot occurs when the system exceeds the setpoint prior to convergence.
- *Low settling time.* Settling time refers to the time required for the controlled system to reach the setpoint.
- *Robustness.* A robust control system converges to the setpoint despite variations in the initial model. This property defines how well the system will react to disturbances and inaccurate measurements.

One advantage of using control systems in self-adaptive software is that the above four properties of a controller can be guaranteed analytically given the mathematical definition of the control system. Thus, a self-adaptive system based on control can provide the user with quantitative guarantees on its convergence, the time to convergence, and its robustness in the face of errors and noise.

Since the advent of autonomic computing [35, 44] and the increasing popularity of self-adaptive software both in research and industry [26, 61], many software controllers adopted the popular feedback loop scheme [10]. However, in most of these cases the similarities with control theory end with the name. Indeed, the use of control theory requires modeling software behavior as a dynamic system; i.e. by means of a system of differential or difference equations.

Abstracting software behavior as a dynamic system is in general a non-trivial task, requiring mathematical skills and expertise not mastered by most software engineers. Model identification methodologies could reduce such difficulty. On the other hand, the availability of a broad set of off-the-shelf identification procedures could lead to accurate models, but with a complexity that makes them quite hard to control [8].

The next section describes our methodology for automatically synthesizing a control system that provides these properties for some non-functional aspect of a software system.

3. CONTROL METHODOLOGY

This section describes our methodology for automatically devising controllers for adaptable software systems. Users provide the initial software system and indicate a tunable parameter, or control variable, that can change the dynamic behavior of the system. Additionally, the users should specify a non-functional requirement for the system to control. Critically, the methodology needs no prior knowledge of the tunable parameter's effects on the specified non-functional aspect. Instead, an appropriate model and control system is automatically devised by the methodology. For example, a user might specify a web service as the software, the number of servers allocated to the service as the tunable parameter, and the response time as the aspect to be controlled. Given these inputs, our methodology will devise a controller that guarantees the desired response by dynamically tuning the server allocation based on measured performance feedback.

The methodology works in two phases, as illustrated in Fig. 1. First, it profiles the software system (the block labeled MB in the figure) to build a model mapping parameter settings into feedback measurements. Second, the methodology uses this model to synthesize a control system (labeled $C(z)$) capable of managing the software's desired non-functional behavior. Different controllers can be synthesized, which trade increasing computational complexity for increased robustness to approximations in the model or unpredictable environmental changes.

3.1 Model Building Phase

Our methodology first builds a model of the system to be controlled. It starts by testing a set of systematically sampled values of the control variable and measuring the effect on the specified non-functional requirement. This process produces a mapping of variable setting to measured feedback. Continuing the web example, model identification measures response time for different numbers of servers.

The model building phase, uses ARPE [53] to build a first order model of the reaction to the control variable. ARPE is based on linear regression and we configure it to identify a model of the form:

$$\mu(k) = \alpha \cdot \eta(k-1) \tag{1}$$

where μ is our measured effect and η is the control variable setting. ARPE determines the value of α, which is then used to synthesize the control system.

The linear model given by (1) may not capture small variations that arise in real systems and it does not deal with abrupt changes in the operating point, like one server becoming unreachable in the platform due to hardware or network failures. However, there are many cases where simple linear models effectively capture a *trend*. For example, increasing the number of servers allocated in our web service example will always speed it up until it reaches the application's maximum parallelism, then the computation speed will not increase. To be effective, the model does not need to capture the exact relationship between the number of servers and the speedup.

To overcome potential errors in the model, our methodology generates three different control systems. Each represents a tradeoff between the computational cost of the controller and its robustness. The first controller has the lowest computational cost and is robust as long as the model captures the trend. The second controller requires more computation, but it updates the identified model online, allowing the system to overcome some unmodeled dynamics and nonlinearities. The third controller is the most computationally expensive but overcomes dramatic errors by deriving a completely new model online instead of incrementally updating it. The first controller is described in Section 3.2 and is used as a baseline to build the second and the third, whose online correction mechanisms are described in Section 3.3. The controller's ability to overcome approximations in the model is discussed in Section 3.4.

3.2 Controller Synthesis Phase

In the second phase, our methodology performs automated control synthesis starting from the model identified in (1). Its goal is to build a control system that tracks the setpoint, rejects disturbances, and tolerates errors in the identified model. We will refer to the setpoint as $\tilde{\mu}$ and the measured feedback at time k as $\mu(k)$. We can measure how well the controller is tracking the setpoint by calculating the error at time k as $e(k) = \tilde{\mu} - \mu(k)$. Small errors indicate the controller is tracking the setpoint well.

A major advantage of control theory is that it provides analytical guarantees about the self-adaptive software system, and thus, achieves predictable behavior in the presence of variability. We perform this analysis using the Z-transform [25], a frequency domain representation of a discrete time control signal (like that in Equation (1)). For example, $C(z)$ represents the Z-transform of the controller,

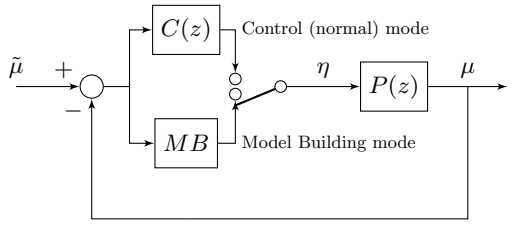


Figure 1: Basic scheme for the first control solution: the control strategy is switched between a first model building phase and a second normal operation mode, where the controller uses the model built in the previous step.

while $P(z)$ represents that of the plant (i.e., the software system). Here, z^{-1} is the unit delay, encoding the temporal shift between the actuation and its effect.

We perform analysis in the Z-domain because doing so makes it easy to prove that the control system has the desired properties. Transforming the discrete-time system into a Z-transform equivalent allows us to use the concept of *transfer functions*, which capture the *input-output* relationship of a function. For example, a controller takes the error signal $e(k)$ as input and outputs a control variable $\eta(k)$. In the Z-domain, the relationship between these two values is expressed as the Z-transform of the output divided by the Z-transform of the input signal; i.e., $C(z) = \mathrm{N}(z)/E(z)$. Our methodology designs the controller by determining the function $C(z)$ that achieves the desired properties. It then performs an inverse transform to produce a set of difference equations that control the behavior of the self-adaptive software. This section describes the process the methodology uses to devise these control equations.

Given a model identified in the previous phase, the methodology first determines the Z-transform of the model given by Equation (1):

$$z \cdot \mathrm{M}(z) = \alpha \cdot \mathrm{N}(z) \tag{2}$$

where $\mathrm{M}(z)$ is the Z-transform of the discrete time feedback signal $\mu(k)$ and $\mathrm{N}(z)$ transforms $\eta(k)$. The control system should select $\eta(k)$ to obtain a certain $\mu(k)$ with the properties described in Section 2 — the guarantee that the system will reach the selected value in a finite time, possibly smaller than a prescribed value and the ability to withstand perturbations and variations.

In order to synthesize the controller — once for all possible systems — we need to express the transfer function $P(z)$ of the plant (software system). Its input is $\mathrm{N}(z)$ while the output is $\mathrm{M}(z)$, therefore $P(z)$ follows from Eq. (2):

$$P(z) = \frac{\mathrm{M}(z)}{\mathrm{N}(z)} = \frac{\alpha}{z}. \tag{3}$$

Our goal is to design a controller, with transfer function $C(z)$ that ensures the desired properties of the self-adaptive software system. The controlled system (the closed-loop feedback system) has a transfer function $G(z)$ which is affected by both the controller and the plant:

$$G(z) = \frac{P(z) \cdot C(z)}{1 + P(z) \cdot C(z)}. \tag{4}$$

Thus, $G(z)$ is an arbitrary transfer function representing the relationship between the setpoint $\tilde{\mathrm{M}}(z)$ and the feedback $\mathrm{M}(z)$. $G(z)$ represents a family of controllers with different tradeoffs between their settling time and ability to track the setpoint while rejecting disturbances. This tradeoff is determined by the *pole* p of $G(z)$; i.e., the value for which the function approaches infinity. For stability, we require $0 \leq p < 1$. Larger values of p produce longer settling times and greater disturbance rejections. Thus, our methodology considers transfer functions of the form:

$$G(z) = \frac{1-p}{z-p} \tag{5}$$

Knowing $G(z)$ (from Eq. (5)) and $P(z)$ (from Eq. (3)), we can solve Eq. (4) to find a family of stable controllers that will track the desired set point:

$$C(z) = \frac{(1-p) \cdot z}{(1-z) \cdot \alpha} \tag{6}$$

The input of the controller is the error between $\tilde{\mathrm{M}}(z)$ and $\mathrm{M}(z)$, while its output is $\mathrm{N}(z)$.

$$C(z) = \frac{\mathrm{N}(z)}{\tilde{\mathrm{M}}(z) - \mathrm{M}(z)} = \frac{(1-p) \cdot z}{(1-z) \cdot \alpha}. \tag{7}$$

Given, this Z-transform of the controller, the methodology simply performs an inverse Z-transform to convert (7) into a discrete time relationship which can easily be implemented in software. Recalling that $e(k) = \tilde{\mu}(k) - \mu(k)$, the methodology synthesizes the controller as:

$$\eta(k+1) = \eta(k) - \frac{1-p}{\alpha} \cdot e(k+1) \tag{8}$$

This equation selects the value of the control variable based on its previous value and on the error between the desired effect and its measured value.

The formal assessment of the properties that this control strategy offers are described in Section 3.4.

3.3 Online Model Updates

This section discusses the three separate mechanisms our methodology uses to provide robustness despite approximations in the models. Each technique uses a different strategy to *update* the model dynamically in response to system variations or model errors. The three techniques are 1) implicit model updating, 2) incremental explicit updating, and 3) model rebuilding. Each extends the previous one, adding additional computation to achieve increased robustness.

3.3.1 Implicit Model Update

The first technique is to simply apply the controller of Section 3.2 without any explicit update mechanism. This technique can overcome modeling errors, as long as the model captures the general trend of the relationship between the control variable and the measured feedback. This solution requires simply measuring feedback and computing the control variable according to (8). Thus, the computational complexity is simply $O(1)$ with a small constant factor.

Remarkably, this simple system is robust, even for extremely noisy applications as long as there are no drastic changes in the trend represented in the model. Our empirical evaluations shows one case study where this controller achieves good results despite a noisy application domain.

3.3.2 Explicit Incremental Update

The second technique works when the online variations are relatively small; i.e., if the application enters a new phase with a different computational load. In this case, the methodology continuously updates an estimate of α (see (8)) while the controller executes. Using this technique, our methodology produces an *adaptive* controller [4] that automatically adjusts itself to accomplish its mission despite changes in the system's dynamics.

The value of α is estimated using a Kalman filter [21, 69]. α's initial value is the result of the linear regression applied during the model building phase. In order to track possible changes in its value, the methodology assumes α varies slowly. The observations coming from data collected online are usually noisy, because of possible intrinsic randomness in the involved phenomena. Assuming the noise is Gaussian with variance q, the resulting dynamic model used for the Kalman filter is:

$$\begin{cases} a_s(k+1) &= \dfrac{\eta(k)-\eta(k-1)}{\mu(k)-\mu(k-1)} + \omega \\ \hat{\alpha}(k) &= a_s(k) \end{cases} \tag{9}$$

where $\omega \sim \mathcal{N}(0, q)$. Using the Kalman filter, the system estimates the model's slope at time k as $\hat{\alpha}(k)$. This estimate of *alpha* is then substituted into (8), so that the controller acts with the most recent update to the model. Computing the Kalman filter updates is still constant time, in terms of complexity, but with a larger constant factor because the Kalman filter must be updated at every time step.

Incremental model updates provide robustness despite shifts and variations in the system, allowing control to be applied even when no single model can capture all dynamics of the deployed system. This approach also allows the controller's linear model to capture unmodeled non-linearities by constantly updating the slope α of the model at the current operating point. This update process is analogous to approximating a curve with a series of tangent lines.

3.3.3 Model Rebuilding

The third technique provides the greatest robustness at a cost of the greatest complexity. Consequently, this technique can handle abrupt variations, like a server failure or other catastrophic change in the system. This technique augments the incremental update process by adding a change point detection procedure to identify when an abrupt change in the environmental conditions requires to restarts the estimation procedure described in Section 3.1.

Specifically, the methodology considers a time window of n control actions. It then computes the average error e_1 for the first $n/2$ samples and the average error e_2 for the second $n/2$ samples. If $|e_1 - e_2| > \mathit{threshold}$ then the rebuilding phase is triggered. This technique prevents triggering a rebuild when the goal is infeasible. This choice, despite its simplicity, is general enough to work in almost all cases. Regardless, this component is modular, permitting a different strategy to be substituted for specific situations. The discussion of the best change point detection technique is outside the scope of this work. Rebuilding the model requires sampling the control variable at different operating points, resulting in a complexity of $O(N)$, where N is the number of possible settings for the variable.

Building a new model from scratch when the error is high ensures that the control system accounts for the current system dynamics. Notice that with a reasonably long time horizon, the incremental updates will eventually converge to the same operating point as the rebuilt model, but we add this safety feature to increase the convergence speed in the face of drastic, unpredicted environmental fluctuations. If the error stays within the acceptable bound, the rebuilding will never be triggered, and the overhead will not be incurred.

3.4 Formal Assessment

Assessing the properties of the controlled system is mainly a matter of checking the model and controller equations. Recalling Section 2, we would like to check that the system is stable, has a low settling time, does not overshoot, and is robust to model inaccuracies. This analysis can largely be performed in the Z-domain.

The Z-transform that represents the controlled system is given by Equation (5). The first three properties can be ensured directly from this equation. Enforcing the stability of the controlled system means ensuring that the pole p is non-negative and less than 1; i.e., $0 \leq p < 1$. Therefore, our methodology will only emit poles in this valid range.

The settling time of the controlled system is also determined from Equation (5). Its inverse transform is

$$\mu(k) = \tilde{\mu} \cdot (1 - p^k) \tag{10}$$

Thus, as k increases the system approaches $\tilde{\mu}$. We define the settling time as the time it takes the system to achieve $(100-\epsilon)\%$ of the final value of $\tilde{\mu}$; i.e., the system's operating point is only a small distance from the desired goal. We refer to this region, which is within ϵ of the goal, as the ϵ confidence zone. Analyzing Equation (10), the first value of k for which our output enters the ϵ confidence zone is

$$k_\epsilon = \frac{\log 0.01\epsilon}{\log |p|} \tag{11}$$

which means that after k_ϵ control steps the signal reaches the confidence zone. That value depends on ϵ, which is usually chosen to be 5%, defining the confidence zone as the interval in which the controlled variable has reached 95% of its final value. In that case $k_\epsilon = \log 0.05 / \log |p|$, which depends only on p. Therefore, the position of the pole determines also how fast the system will reach its equilibrium.

The pole's value p can be used to trade responsiveness — how fast the controller reacts, measured as settling time — and robustness in the face of noise or unmodeled variance in system behavior. The controller acts based on its model, or estimation of the effect of its action on the system. As noted above, even the simplest formulation of the controller can overcome errors in the model because the system actively incorporates feedback, which keeps the controller informed of the effect of its action.

To complete the tradeoff analysis, we show the relationship of the pole p to the error the system can withstand. Assume our methodology estimates $\alpha(k)$ as $\tilde{\alpha}(k)$, but the true value is $\tilde{\alpha}(k) \cdot \Delta(k)$. This multiplicative perturbation is often used to quantify the error of an estimation. For example, $\Delta = 10$ implies that the estimate may be 10 times smaller or larger than the true value.

We test the largest perturbation that our system can withstand while still tracking the setpoint. In other words, we want to find the values of $\Delta(k)$ for which our plant is still stable. The plant transfer function $P(z)$ is $\frac{\alpha}{z}$, therefore, given a perturbation Δ, it becomes $P_\Delta(z) = \frac{\tilde{\alpha} \cdot \Delta}{z}$. The con-

troller transfer function is $C(z) = \frac{(1-p)\cdot z}{(z-1)\tilde{\alpha}}$. The controlled system's transfer function under perturbation becomes:

$$G_\Delta = \frac{C(z)\cdot P_\Delta(z)}{1 + C(z)\cdot P_\Delta(z)} = \frac{(1-p)\cdot\Delta}{z + \Delta(1-p) - 1} \quad (12)$$

which is, again, stable and without oscillation if and only if the two denominator poles are between 0 and 1.

Thus for a stable system, $0 < \Delta(k) < \frac{2}{1-p}$, which means that choosing the value of the pole p defines how safely the controller acts with respect to model perturbations. If p is 0.1 the estimation can be inaccurate by a factor of 2. In conclusion, there is a fundamental tradeoff between the controller reactivity and the safety with respect to perturbations that the controller can withstand. This tradeoff can be exploited carefully choosing the pole p.

This relationship between the pole and the tolerable perturbation quantifies what we mean when we say the basic control system will provide implicit model updating as long as the model captures the trend between the control variable and the feedback signal. For example, for a pole of $p = 0.9$, the system can tolerate a perturbation of 20. Suppose the methodology provides a model that estimates speedup as a function of allocated servers. The predicted speedup can be off by a factor of 20 without affecting setpoint tracking and stability. The additional techniques of incremental model updating and model rebuilding allow the controlled system to update the model and provide these same guarantees in the face of unmodeled non-linearities (that exceed the tolerable perturbation) or catastrophic system failures that invalidate the model completely. We note that the methodology outputs control systems providing these guarantees despite the fact that it begins with no a priori knowledge of the control variable.

4. EXPERIMENTAL EVALUATION

In this section we describe our experimental evaluation. We have three different case studies, each demonstrating one of the three strategies for dynamically updating the controller's model. We evaluate the first two results by means of two common metrics, the Mean Square Error (MSE) and the Mean Average Percentage Error (MAPE). Recalling that $\tilde{\mu}$ represents the goal and μ its actual value, the MSE and MAPE are defined as

$$\begin{aligned} MSE &= \frac{1}{n}\sum_{i=1}^{n}[\tilde{\mu} - \mu(i)]^2 \\ MAPE &= \frac{1}{n}\sum_{i=1}^{n}\left|\frac{\tilde{\mu} - \mu(i)}{\mu(i)}\right| \end{aligned} \quad . \quad (13)$$

Clearly, the MSE and MAPE represent two different metrics on the system. The MSE is related to the quantity involved, in the sense that if the signals have a low magnitude, the MSE can be small but still represent significant errors. On the contrary, the MAPE is a relative number that determines how far from optimal the system is. In the third test case we will show some images of the controller behavior, to make it easy to visibly grasp the benefits of using our methodology to automatically devise control strategies.

4.1 Video Compression

The first case study deals with video compression. We suppose that a camera is recording a video to be streamed over the network and stored in an archive. Our case study started thinking of surveillance video but this is not the only example that we can come up with to justify our choice. For example, if you have limited channel to stream news video like BBC or CNN you might want to do something similar.

We divide the video into frames — using one jpg for each frame — and send the frame separately over the network. The frames can be preprocessed to reduce the quality of the image as much as possible, maintaining however some acceptable standards on the resulting quality. The quality loss should enable us to reduce the size of the compressed image and therefore the disk space needed to store the frames. Our primary aim is the image quality reduction that should follow a certain setpoint.

In this case, our software system is the video encoder. Our non-functional requirement is the quality of the compressed videos quantified as structural similarity (SSIM) index [68]. SSIM is a unitless metric that ranges from 0 to 1, with values closer to 1 indicating images that are very close. We use SSIM to quantify the quality loss due to compression. In this example, our control variable is a command line parameter that indicates the density of the compression procedure. Given these inputs, we use our methodology to automatically devise a control system that selects the density parameter for the next frame based on the measured SSIM score for the previous frame.

We evaluate our methodology by running the compression scheme with the synthesized controller for a number of videos. For each video we compute the average compression percentage $\mathbf{c}_\%$ per frame, the average SSIM μ, the MSE and MAPE. The data are reported in Table 1. As can be seen, with all the videos we achieve a reduction in space that is superior to 75%, reducing the quality. Our μ value is sometimes superior to the setpoint, since the image could not have been reduced further without a too big quality loss (for example in the 0.8 SSIM version of pumpkin candle). The procedure is shown to work both with high resolution videos and with low resolution ones. The reduction in size is a consequence of the quality setpoint, but it can be noted that the controller invariably achieves good compression and setpoint tracking. In fact the MSE and MAPE values are uniformly low.

4.2 Energy Control

Our second case study also deals with video compression; however, to demonstrate the generality of the approach, we now control energy instead of quality. This can be useful to extend battery life when encoding video on a mobile device, or to save energy bills when working on servers. We can influence the energy of the system by changing the speed of the processor. Recent processors support tradeoffs between the processor speed and energy, allowing the system to perform more work for a greater energy consumption [32]. We would like to use our methodology to synthesize a controller which will maintain energy goals.

Again, our software system is the video encoder. Our non-functional requirement is now the energy consumption of the video encoder running on our Intel Xeon dual-socket E5-2690 system. The system is connected to a Wattsup device that provides real-time feedback of full-system energy consumption. In this case we measure energy consumption relative to the default configuration with the processors at full speed. In this example, our control variable is the processor's clock speed, which is available to software through

Table 1: Video compression through density reduction results. The resulting quality μ is close to the specified goal. Thanks to the compression, there is an evident reduction in the video size.

video	frames	resolution	goal	μ	MSE	MAPE	size_o	size_c	$\text{c}_\%$
obama victory speech	17118	480×270	0.8	0.819	0.00057	2.55%	243.2 Mb	48.5 Mb	-80.1%
			0.9	0.923	0.00069	2.66%		95.8 Mb	-60.6%
samsung advertisment	2112	1920×1080	0.8	0.873	0.01345	12.11%	172.2 Mb	34.6 Mb	-79.9%
			0.9	0.933	0.00248	12.66%		50.2 Mb	-70.8%
amazing nature	7131	1920×1080	0.8	0.842	0.00619	6.76%	804.0 Mb	141.1 Mb	-82.4%
			0.9	0.926	0.00144	3.30%		230.9 Mb	-71.3%
planet earth from space	36997	1920×1080	0.8	0.832	0.00436	6.21%	2408.1 Mb	417.3 Mb	-82.7%
			0.9	0.924	0.00109	3.08%		834.1 Mb	-65.4%
lawnmower	1904	1920×1080	0.8	0.817	0.00044	2.24%	1752.4 Mb	50.4 Mb	-97.1%
			0.9	0.915	0.00026	1.69%		80.2 Mb	-95.4%
night traffic	312	1920×1080	0.8	0.815	0.00798	9.41%	138.2 Mb	8.0 Mb	-94.2%
			0.9	0.912	0.00157	3.51%		8.8 Mb	-93.6%
new york traffic	182	1920×1080	0.8	0.834	0.00321	4.31%	201.8 Mb	21.1 Mb	-89.5%
			0.9	0.935	0.00142	3.85%		24.2 Mb	-88.0%
pumpkin candle	385	1920×1080	0.8	0.898	0.00974	12.30%	250.2 Mb	10.2 Mb	-95.9%
			0.9	0.908	0.00027	12.20%		10.2 Mb	-95.9%
raining	1768	1920×1080	0.8	0.853	0.00287	6.66%	1122.7 Mb	32.2 Mb	-97.1%
			0.9	0.912	0.00018	1.40%		42.4 Mb	-96.2%
speedometer	478	1920×1080	0.8	0.822	0.00736	10.14%	201.5 Mb	15.6 Mb	-92.3%
			0.9	0.911	0.00033	1.25%		20.2 Mb	-90.0%
alpha centaury	1754	1920×1080	0.8	0.835	0.00976	7.51%	129.7 Mb	4.2 Mb	-96.7%
			0.9	0.921	0.00079	2.27%		8.0 Mb	-93.9%

the `cpufrequtils` package in Linux. With these inputs, we use our methodology to automatically devise a control system that selects the processor speed for the next frame based on the measured energy consumption of the previous frame.

We evaluate our methodology by running the encoder with the synthesized controller for several videos. For each video, we set two targets: the first is an energy efficiency improvement of 5% (1.05×) and the second is 10% (1.10×). We use our methodology to automatically generate one controller that uses implicit updating and one that does incremental updating with the Kalman filter. For each video, we run each controller for each energy goal and report the results in Table 2, which shows the average energy efficiency gain (μ), as well as the MSE and MAPE. As shown in the table, all videos achieve average energy consumption very close to the desired value, with uniformly low MSE and MAPE. While both controllers provide good results, the Kalman filer tends to have lower error. We note that these energy efficiencies include the overhead of running the controller itself.

4.3 Service Dynamic Binding

In the context of Service Oriented Architectures (SOA) dynamic binding is the mechanism allowing abstract operations to be mapped to concrete components implementing them. If multiple functionally equivalent implementations are available for an abstract operation, the selection among them is usually based on their non-functional properties (e.g. in [3, 7, 12, 24, 39, 70, 71]). Consider for example a software providing a geo-localization service that has to assure a certain reliability per request $\tilde{\mu}$. This means that whenever a request is issued, the service can fail to serve it with a probability of at most $1 - \tilde{\mu}$ [23, 24]. Assume the abstract localization operation could be backed either on the services $S1$ Maps or $S2$ Maps. The availability of each of these two alternatives may change at runtime, for example because of changing load conditions, network timeouts, or maintenance. The goal of the dynamic binder is to decide, for each incoming request, which alternative to select in order to continuously provide the desired reliability $\tilde{\mu}$ in spite of possible changes in the reliabilities of $S1$ and $S2$.

In [24], we faced this problem by manually modeling the behavior of the system through a discrete time Markov chain and devising a suitable controller by hand. In this work we consider the software as a black-box and let our methodology automatically construct a model of it and generate a suitable controller[1].

We assume a monitoring infrastructure estimating at each time point k the actual reliability $\mu(k)$ of our geo-localization service and feeding its current value to the controller. The error is then quantified as the difference between the monitored reliability and its target setpoint ($\tilde{\mu} - \mu(k)$). The goal of the controller is to decide the value of the control variable η, which represents the probability of selecting $S1$, in order to keep the error as close as possible to 0 (consequently, the probability of selecting $S2$ is $1 - \eta$).

Our experiments are reported in Figure 2. All plots show the initial model building phase (emphasized by a different background pattern) when the domain of the control variable is explored in order to approximate its relation with

[1]As explained in [24], having a module able to automatically decide the dynamic binding problem between two alternatives is enough to solve the multiple alternative problem by composing multiple instances of it into a binary decision tree.

Table 2: Results Controlling Energy Consumption.

video	frames	resolution	goal	controller	μ	MSE	MAPE
blue_sky_1080p25	200	1920×1080	1.05	implicit update	1.049877	0.002274	3.6847%
			1.05	kalman	1.050014	0.001555	2.9060%
			1.10	implicit update	1.099538	0.001614	2.8696%
			1.10	kalman	1.099577	0.001690	2.9043%
crowd_run_1080p	500	1920×1080	1.05	implicit update	1.049314	0.000834	2.0883%
			1.05	kalman	1.049956	0.000761	2.0075%
			1.10	implicit update	1.099038	0.003235	4.3388%
			1.10	kalman	1.099530	0.000876	2.0423%
ducks_take_off_1080p	500	1920×1080	1.05	implicit update	1.049401	0.000607	1.7965%
			1.05	kalman	1.049973	0.000566	1.6930%
			1.10	implicit update	1.099134	0.002211	2.8744%
			1.10	kalman	1.099550	0.000697	1.7568%
factory	500	1920×1080	1.05	implicit update	1.049373	0.001951	3.2981%
			1.05	kalman	1.049993	0.001959	3.2854%
			1.10	implicit update	1.099119	0.002927	3.9282%
			1.10	kalman	1.099566	0.002145	3.2917%
in_to_tree_1080p	500	1920×1080	1.05	implicit update	1.049288	0.001302	2.6284%
			1.05	kalman	1.049909	0.001128	2.4571%
			1.10	implicit update	1.099055	0.001951	3.0843%
			1.10	kalman	1.099484	0.001256	2.4589%
old_town_cross_1080p	500	1920×1080	1.05	implicit update	1.049187	0.002158	3.5053%
			1.05	kalman	1.049956	0.001468	2.8307%
			1.10	implicit update	1.098879	0.001809	3.0349%
			1.10	kalman	1.099529	0.001647	2.8690%
pedestrian_area	500	1920×1080	1.05	implicit update	1.050360	0.003810	4.7949%
			1.05	kalman	1.050268	0.002746	3.8205%
			1.10	implicit update	1.099219	0.004492	5.0264%
			1.10	kalman	1.099828	0.002946	3.8146%
station2	500	1920×1080	1.05	implicit update	1.049169	0.005904	5.8861%
			1.05	kalman	1.050097	0.004919	5.2071%
			1.10	implicit update	1.097553	0.005493	5.3570%
			1.10	kalman	1.093600	0.004564	4.9187%
sunflower	500	1920×1080	1.05	implicit update	1.049265	0.003481	4.4971%
			1.05	kalman	1.049995	0.002960	4.0613%
			1.10	implicit update	1.099033	0.003673	4.4298%
			1.10	kalman	1.099534	0.003193	4.0465%
tractor	500	1920×1080	1.05	implicit update	1.049481	0.001545	2.9352%
			1.05	kalman	1.050077	0.001065	2.3507%
			1.10	implicit update	1.099305	0.002122	3.3338%
			1.10	kalman	1.099645	0.001208	2.4095%

the measured reliability. Figures 2a and 2b show the step response of two automatically generated controllers. The first was configured to have a pole in 0.9, while the second in 0.3. As expected from the discussion of Section 3, both of the controllers are stable and converge to the setpoint, when feasible. The former has a longer settling time than the latter after a change in the setpoint occurs. Although, when settled, the two controllers show a coherent behavior, leading exactly to the same steady-state performance. Notice also at time 1300 the goal becomes unfeasible and both of the controllers keep trying to minimize the error, moving the actual reliability *as close as possible* to the setpoint. Figures 2c and 2d replicates the same scenario of 2a and 2b, but with the addition of white noise with a standard deviation of 10^{-2} and bounded by $[-0.1, 0.1]$ to each of the reliabilities of $S1$ and $S2$. Despite its longer settling time, the controller with pole in 0.9 shows a significantly more effective rejection of the disturbances than the controller with pole in 0.3. The former is also more effective in avoiding overshooting the control signals, while the latter introduces many spikes, temporarily compromising the stability of the controlled system. This is an empirical example of trade-off between short settling time and robustness to noise that has been discussed in Section 3, which requires a careful decision about the position of the pole. Figure 2e, shows the situation where the reliability of $S1$ changes smoothly. Thanks to the continuous tracking of α enacted by the Kalman filter, the model of the system is updated online, allowing the controller to cope with the variations without undergoing a new identification phase. Finally, in Figure 2f the case of an abrupt change in the reliabilities of $S1$ and $S2$ is reported. The reliabilities of the two services sharply change at time 1000. After a short time, the change detection mechanism detects the change and triggers a model rebuilding. The re-

sulting updated model captures the new situation, allowing for effective control.

A Matlab implementation of this case study is available online[2] for simulation purposes. Numerical mathematical programming is an established instrument for control experts to study controller's performance by simulating disturbances and process dynamics. A Java prototype implementation, based on the Spring Framework [40], can be downloaded. Spring is an industrial strength lightweight container for J2EE applications. The monitoring infrastructure and the controller have been implemented through the Spring Aspect Oriented Programming (AOP) features to show how our methodology can be integrated in existing applications without an unfeasible burden on the established development cycle (for further details on the Spring framework, please refer to [40] or the official website[3]). A running instance of the Spring implementation is also accessible from previous url, with a web-interface to ease the demonstration.

5. RELATED WORK

Adaptation is becoming a key concern in software applications [14, 46]. An adaptive application must select from many configurations the one that is most appropriate to obtain some specific performance result. There are many examples, from hardware to software development. The evaluation of a new microprocessor design requires studying the impact of input data sets and workload composition [20]. Compiler-level advancements have been developed to support adaptive implementations for performance [2, 33, 43, 66] or power [6, 62], and low level architectures are dynamically adjusted and targeted [9, 38, 48, 63]. Another example comes from High Performance Computing, where it is common to change an application parameter to adapt a running application. In [41] a threshold value is changed while executing parallel Monte Carlo ocean color simulations, while [18] presents a study on tuning Fast Fourier Transformations on graphic processing units. Also, Rahman et al. [58] and Tiwari et al. [67] studied the effect of compiler parameters on both performance and power/energy consumption for scientific computing. A lot of modeling and tuning effort has recently been devoted to the specific application of MapReduce [5, 15, 30, 50, 59, 60].

Self-management techniques are also prominent in industry; e.g., companies like IBM [36] (see projects like the IBM Touchpoint Simulator, the K42 Operating System [47]), Oracle (Oracle Automatic Workload Repository [56]), and Intel (Intel RAS Technologies for Enterprise [37]).

Control theory [27, 28] is capturing an increasing interest from the software engineering community that looks at self-management as a means to meet QoS requirements despite unpredictable changes of the execution environment [57]. Examples of this trend can be seen in research on control of web servers [45, 51], data centers and clusters management [19, 49], operating systems [13, 34, 42, 47, 52, 54, 55], and across the system stack [31].

The application of control theory in software engineering, however, is still in a very preliminary stage. Developing accurate system models for software is in fact hard. Moreover, strong mathematical skills are needed in order to deal with complex non-linear dynamics of real systems [17, 28, 72]. These difficulties usually lead to the design of controllers focused on particular operating regions or conditions and ad hoc solutions that address a specific computing problem using control theory, but do not generalize [51, 64, 65]. For example, in [29] the specific problem of building a controller for a .NET thread pool is addressed.

This work aims at leveraging the effort of adopting formally guaranteed control theory methods for software adaptation by providing a widely applicable push-button methodology, which reduces the need for strong mathematical background to devise ad-hoc modeling and control solutions.

Despite their broad variety, most of the methodologies for the development of self-adaptive applications resembles a three steps process: data collection and analysis, modeling, and control. The glue of the whole process is usually the software model. It has indeed the purpose of filling the gap between the possible control choices and the effect they have on the satisfaction of the requirements. This is not a novelty of self-adaptive software, but its advent is casting a new light on the property such model has to satisfy: while complex, precise quality models have been used in the past to enable design-time optimization of software architectures [1], such complexity often inhibits their applicability for runtime adaptation because of the short time available for verification and control [22].

Our methodology proposes a systematic data collection procedure enabling the automatic construction of an approximate model. Once built through a first identification phase, such model can easily be kept updated online and allows for the application of established and effective control results.

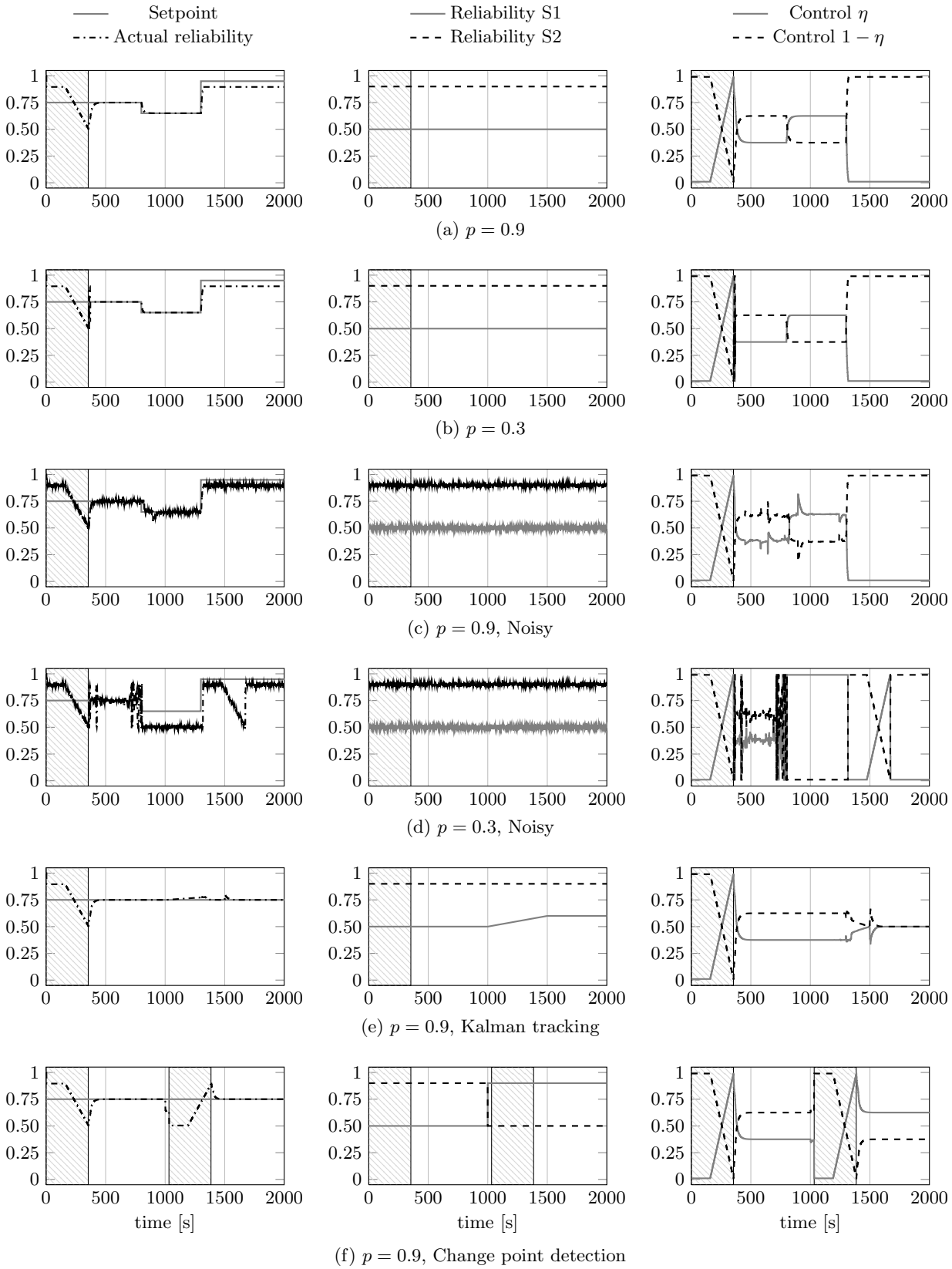
6. CONCLUSION AND FUTURE WORK

This paper proposes a methodology that takes a tunable variable and feedback mechanism to produce a closed-loop control strategy that provides formal guarantees for an adaptive software system's dynamic behavior. The methodology leverages control theory to prove that whenever the desired behavior is feasible, our controller is capable of selecting the appropriate variable setting to achieve the goal, despite unpredicted disturbances and approximate estimates and measurements. Our results show that it is possible to automatically design a controller that trades off responsiveness and robustness to model perturbations. This methodology makes it possible for non-experts to build self-adaptive software that benefits from mathematically grounded control theoretic techniques and their formal guarantees without the background needed to design ad-hoc solutions, which might however be needed to improve the controller performance on specific problems.

Our system would work also in case of multiple independent control variables and output measures, however we plan to study an automatic control synthesis technique for coupled Multiple Input Multiple Output (MIMO) systems; i.e., when the controller needs to coordinate several control variables toward the satisfaction of multiple, possibly conflicting, objective requirements. We also plan to incorporate the synthesis of multiple different controllers and to switch between those strategies at runtime, allowing to continuously adjust the trade off between responsiveness and robustness. In addition, we plan to exploit the capabilities of the Kalman filter to forecast future values of α and define a proactive control strategy, to support complex adaptation strategies.

[2] `http://www.iste.uni-stuttgart.de/rss/people/filieri/icse-2014-control`

[3] `http://www.springsource.org`



(a) $p = 0.9$

(b) $p = 0.3$

(c) $p = 0.9$, Noisy

(d) $p = 0.3$, Noisy

(e) $p = 0.9$, Kalman tracking

(f) $p = 0.9$, Change point detection

Figure 2: Behavior of the dynamic binding system in different conditions and for different configurations.

7. REFERENCES

[1] A. Aleti et al. "Software Architecture Optimization Methods: A Systematic Literature Review". In: *Software Engineering, IEEE Transactions on* 39.5 (2013), pp. 658–683.

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

[3] D. Ardagna and R. Mirandola. "Per-flow optimal service selection for Web services based processes"". In: *Journal of Systems and Software* 83.8 (2010), pp. 1512 –1523.

[4] K. J. Astrom and B. Wittenmark. *Adaptive Control*. 2nd. 1994. ISBN: 0201558661.

[5] S. Babu. "Towards automatic optimization of MapReduce programs". In: *SoCC*. 2010, pp. 137–142. ISBN: 978-1-4503-0036-0.

[6] W. Baek and T. Chilimbi. "Green: A Framework for Supporting Energy-Conscious Programming using Controlled Approximation". In: *ACM PLDI*. 2010.

[7] N. Ben Mabrouk et al. "QoS-Aware Service Composition in Dynamic Service Oriented Environments". In: *Middleware*. Vol. 5896. Lecture Notes in Computer Science. 2009, pp. 123–142.

[8] S. Billings. "Identification of nonlinear systems: a survey". In: *CTA* 127.6 (1980), pp. 272–285.

[9] R. Bitirgen et al. "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach". In: *MICRO*. 2008, pp. 318–329.

[10] Y. Brun et al. "Engineering Self-Adaptive Systems through Feedback Loops". In: *Software Engineering for Self-Adaptive Systems*. 2009, pp. 48–70.

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

[12] G. Canfora et al. "An approach for QoS-aware service composition based on genetic algorithms". In: *GECCO*. 2005, pp. 1069–1075.

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

[14] B. Cheng et al. "Software Engineering for Self-Adaptive Systems: A Research Roadmap". In: *Software Engineering for Self-Adaptive Systems*. 2009, pp. 1–26.

[15] J. Dean and S. Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113.

[16] Y. Diao et al. "Self-managing systems: a control theory foundation". In: *ECBS Workshop*. 2005, pp. 441–448.

[17] R. Dorf and R. Bishop. *Modern control systems*. Prentice Hall, 2008. ISBN: 0132270285.

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

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

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

[21] G. Evensen. *Data assimilation: the ensemble Kalman filter*. Springer, 2009.

[22] A. Filieri et al. "Run-time efficient probabilistic model checking". In: *ICSE*. ACM, 2011, pp. 341–350.

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

[24] A. Filieri et al. "Reliability-driven dynamic binding via feedback control". In: *SEAMS*. June 2012.

[25] G. F. Franklin et al. *Feedback Control of Dynamic Systems*. 2009.

[26] W. Gentzsch et al. "Self-Adaptable Autonomic Computing Systems: An Industry View". In: *Proceedings of the 16th International Workshop on Database and Expert Systems Applications*. 2005, pp. 201–205.

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

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

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

[30] H. Herodotou and S. Babu. "Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs". In: *PVLDB* 4.11 (2011), pp. 1111–1122.

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

[32] H. Hoffmann. "Racing and Pacing: An Evaluation of Heuristics for Energy-aware Resource Allocation". In: *Hot Power*. 2013.

[33] H. Hoffmann et al. "Dynamic Knobs for Responsive Power-Aware Computing". In: *ASPLOS*. 2011.

[34] H. Hoffmann et al. "A Generalized Software Framework for Accurate and Efficient Management of Performance Goals". In: *EMSOFT*. 2013.

[35] M. C. Huebscher and J. A. McCann. "A survey of autonomic computing: degrees, models, and applications". In: *ACM Comput. Surv.* 40.3 (Aug. 2008), 7:1–7:28.

[36] IBM Inc. *IBM Autonomic Computing website*. `http://www.research.ibm.com/autonomic/`. 2009.

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

[38] E. Ipek et al. "Core fusion: accommodating software diversity in chip multiprocessors". In: *SIGARCH Comput. Archit. News* 35.2 (2007), pp. 186–197.

[39] M. Jaeger et al. "QoS-Aware Composition of Web Services: An Evaluation of Selection Algorithms". In: *On the Move to Meaningful Internet Systems*. Vol. 3760. 2005, pp. 646–661.

[40] R. Johnson et al. *Professional Java Development with the Spring Framework*. 2005.

[41] T. Kajiyama et al. "Statistical performance tuning of parallel Monte Carlo ocean color simulations". In: *PDCAT*. Dec. 2012, pp. 761–766.

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

[43] T. Karcher and V. Pankratius. "Run-time automatic performance tuning for multicore applications". In: *EUROPAR*. 2011, pp. 3–14.

[44] J. O. Kephart and D. M. Chess. "The Vision of Autonomic Computing". In: *Computer* 36.1 (Jan. 2003), pp. 41–50.

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

[46] J. Kramer and J. Magee. "Self-Managed Systems: an Architectural Challenge". In: *FOSE*. 2007, pp. 259 – 268.

[47] O. Krieger et al. "K42: Building a Complete Operating System". In: *EuroSys*. 2006.

[48] R. Kumar et al. "Processor Power Reduction Via Single-ISA Heterogeneous Multi-Core Architectures". In: *Computer Architecture Letters* 2.1 (2003), p. 2.

[49] D. Kusic and N. Kandasamy. "Risk-aware limited lookahead control for dynamic resource provisioning in enterprise computing systems". In: *Cluster Computing* 10 (4 2007), pp. 395–408.

[50] J. Liu et al. "Panacea: towards holistic optimization of MapReduce applications". In: *CGO*. 2012, pp. 33–43.

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

[52] M. Maggio et al. "Power Optimization in Embedded Systems via Feedback Control of Resource Allocation". In: *Control Systems Technology, IEEE Transactions on* 21.1 (2013), pp. 239–246. ISSN: 1063-6536. DOI: 10.1109/TCST.2011.2177499.

[53] M. Maggio and H. Hoffmann. "ARPE: A Tool To Build Equation Models of Computing Systems". In: *Feedback Computing 2013*. 2013.

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

[55] S. Oberthür et al. "Dynamic online reconfiguration for customizable and self-optimizing operating systems". In: *EMSOFT*. 2005, pp. 335–338.

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

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

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

[59] N. Rizvandi et al. "On Using Pattern Matching Algorithms in MapReduce Applications". In: *ISPA*. 2011, pp. 75–80.

[60] N. B. Rizvandi et al. "On Modelling and Prediction of Total CPU Usage for Applications in MapReduce Environments". In: *ICA3PP*. 2012, pp. 414–427.

[61] M. Salehie and L. Tahvildari. "Self-adaptive software: Landscape and research challenges". In: *ACM Trans. Auton. Adapt. Syst.* 4.2 (2009), pp. 1–42.

[62] J. Sorber et al. "Eon: a language and runtime system for perpetual systems". In: *SenSys*. 2007, pp. 161–174.

[63] M. A. Suleman et al. "Accelerating critical section execution with asymmetric multi-core architectures". In: *ASPLOS*. 2009, pp. 253–264.

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

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

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

[67] A. Tiwari et al. "Auto-tuning for energy usage in scientific applications". In: *Euro-Par*. 2012, pp. 178–187.

[68] Z. Wang et al. "Image quality assessment: from error visibility to structural similarity". In: *IEEE Transactions on Image Processing* 13.4 (2004), pp. 600–612.

[69] G. Welch and G. Bishop. *An introduction to the Kalman filter*. 1995.

[70] T. Yu et al. "Efficient algorithms for Web services selection with end-to-end QoS constraints". In: *ACM Transactions on the Web* 1 (1 2007).

[71] L. Zeng et al. "QoS-aware middleware for Web services composition". In: *IEEE Transactions on Software Engineering* 30.5 (2004), pp. 311–327.

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