Software Engineering Meets Control Theory

Antonio Filieri*, Martina Maggio*
Konstantinos Angelopoulos, Nicolás D'Ippolito, Ilias Gerostathopoulos, Andreas Berndt Hempel,
Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, Filip Krikava,
Sasa Misailovic, Alessandro Vittorio Papadopoulos, Suprio Ray, Amir M Sharifloo,
Stepan Shevtsov, Mateusz Ujma, Thomas Vogel

*Dagstuhl Seminar Organizer

***Abstract*—The software engineering community has proposed numerous approaches for making software self-adaptive. These approaches take inspiration from machine learning and control theory, constructing software that monitors and modifies its own behavior to meet goals. Control theory, in particular, has received considerable attention as it represents a general methodology for creating adaptive systems. Control-theoretical software implementations, however, tend to be ad hoc. While such solutions often work in practice, it is difficult to understand and reason about the desired properties and behavior of the resulting adaptive software and its controller.**

This paper discusses a control design process for software systems which enables automatic analysis and synthesis of a controller that is guaranteed to have the desired properties and behavior. The paper documents the process and illustrates its use in an example that walks through all necessary steps for self-adaptive controller synthesis.

I. Introduction

Software's pervasiveness in every context of life places new challenges on Software Engineering. Highly dynamic environments, rapidly changing requirements, unpredictable and uncertain operating conditions demand new paradigms for software design. Runtime adaptation mechanisms are required to deploy robust software that operates correctly despite a lack of design-time knowledge.

In the last decade, the Software Engineering community developed a multitude of approaches for developing *self-adaptive* software systems [1, 8, 14, 23, 30, 48, 52, 59, 72]. While these approaches describe general *techniques*, their *implementations* focus on specific problems and do not extend to broadly implementable methodologies [25]. Furthermore, many proposed adaptation mechanisms lack the theoretical grounding to ensure their dependability beyond the few specific cases for which they are implemented [34, 51, 67, 68]. In the last century, Control Theory has established a broad family of mathematically grounded techniques for adaptation. Controllers provide formal guarantees about their effectiveness and behavior under precise assumptions on the operating conditions.

Although the similarities between industrial plant control and software adaptation are self-evident, most attempts to apply off-the-shelf control-theoretical results to software systems have achieved limited scope, mostly tailored to specific applications and lacking rigorous formal assessment of the applied control strategies. The two main difficulties in applying control theory to software systems are: (i) developing mathematical models of software behavior suitable for controller design, and (ii) a lack of Software Engineering methodologies for pursuing controllability as a first class concern [16, 20, 37, 74].

A theoretically sound controller requires mathematical models of the software system's dynamics. Software engineers often lack the background needed to develop these models. Analytical abstractions of software established for quality assurance can help fill the gap between software models and dynamical models [19, 27, 71], but they are not sufficient for full control synthesis. In addition, goal formalization and knob identification have to be taken into account to achieve software controllability [57].

Several approaches have extended control-theoretical results into more general methodologies for designing adaptive software systems. Pioneering works in system engineering [1, 17, 35] spotlighted how control theoretical results improve computing system design. These contributions have been especially influential for performance management and resource allocation; however, new trends in self-adaptive software introduce new software models and a variety of quantitative and functional requirements beyond the scope of those works. More recently, methodological approaches for performance control [2, 58] and the design of self-adaptive operating systems [48] have been proposed. Software Engineering and Autonomic Computing has also highlighted the centrality of feedback loops for adaptive systems [8, 41].

Control theory has developed during the years for physical systems, where what to measure and what to control is very clear, as well as how to define physical models of the actors. For these systems, many of the problems have been solved and the resulting solutions have been mathematically grounded. However, the software engineering domain poses different challenges, with respect to the physical world. On one hand, quantities that are difficult to measure in the physical world could be extremely easy to measure instrumenting code. On the other hand, there is no basic physic that one could rely upon and it could be extremely complex to come up with equations that properly describe the running system. This paper casts the control-theoretical framework of time-based controllers into the problem of applying control theory to software systems and provides a *comprehensive analysis of one of the paradigms in the control theoretical design for*

self-adaptive software systems, matching the various phases of software development with the relevant background and techniques from Control Theory. For space limitations we restrict this paper to time-based controllers. Other paradigms, *e.g.* event-based controllers [18, 19], allow achieving similar or complementary results for specific control problems. The goal is to bootstrap the design of mathematically grounded controllers, sensors, and actuators for self-adaptive software systems, providing formally provable effectiveness, efficiency, and robustness.

II. Adaptation in software and control engineering

The word "adaptation" means two different things to software and control engineers. For a software engineer, adaptive software reacts to changing environmental and contextual conditions by changing its behavior or structure. Adaptation, however, is only one of the software designer's concerns. For a control engineer, controlling a system is the only concern – adaptive control just adds another degree of flexibility, where the controller may change its own control policies as well. Even the simplest controller (*e.g.*, one that maintains room temperature by turning a heater on and off) makes the system *adaptive* according to the definition understood by a software engineer. To avoid misunderstandings, this section discusses more precisely the meaning of adaptation in the two communities and subsequently analyzes how the two viewpoints can be reconciled.

From a **software engineering perspective**, a system is adaptive when it allows for modifying its structure or behavior at runtime; *i.e.* without interrupting its service. An adaptive system can be coupled with an adaptation manager to make it continuously satisfy its requirements. A requirement violation may occur during runtime due to changes in execution environment including user interaction, the behavior of third-party components, or because the requirements themselves change. Coupling the system with its adaptation manager creates what is called a *self-adaptive* system.

Self-adaptive systems have been an aim of Software Engineering for about two decades. However, general frameworks have been proposed only since the late 90s, both in the Software Engineering community and on the new field of Autonomic Computing [40, 41, 55, 62]. While each framework is unique, all of them share a *closed-loop* structure where the software monitors requirements violations, plans counteractions, and enforces them.

To ground the concepts, we take as reference the most popular self-adaptive design framework: the Monitor-Analyze-Plan-Execute (MAPE) feedback loop (Figure 1) [41]. The adaptive system operates in a changing environment, affecting its ability to satisfy the requirements. The adaptation manager detects the changes, analyzes their impact, and, if needed, plans and executes actions in response to the changes. All phases can be supported by additional knowledge about the system; *e.g.*, suitable models kept updated at runtime.

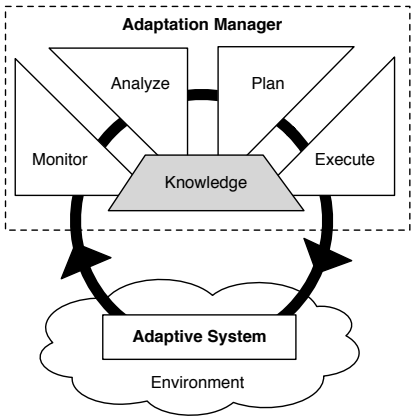

Fig. 1: Adaptive system: the *software engineering perspective*.

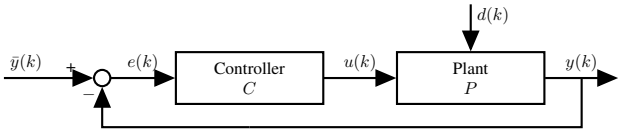

Fig. 2: Adaptive system: the *control perspective*.

From the **control engineering perspective**, an adaptive system consists of a control loop (Figure 2 shows an example) [3, 35]. A closed loop system consists of a controller and a plant denoted as C and P, respectively. The time is assumed to be discretized and $y(k)$ represents the value of the signal y at time instant k. The controller's input is the error $e(k)$ between the goals $\bar{y}(k)$ and the measured value of the outputs $y(k)$ – where input and output are measurable quantities[1]. The controller's output $u(k)$ is a vector of values for the configurable parameters of the plant, often called *knobs* or *actuators*. All inputs and outputs can be vectors, meaning that the measured output of the closed loop system can be, for example, a two-dimensional vector whose elements are reliability and response times. The number of goals $\bar{y}(k)$ should be equal to the number of measured outputs, while the number of knobs for which the controller should compute a value can differ from the dimensionality of the output. When control is applied to software systems, the plant is usually identified with the adaptive software system under control, but it can also contain additional information about the execution environment and platform. A plant receives as input the knob configurations and produces as output some measures of quantities $y(k)$ upon which the user sets the control goals. There are other entities that act on the behavior of the plant, other than the control variables $u(k)$. For example, if $y(k)$ is the execution speed of a loop in the software, the sudden start of the garbage collector can slow down the computation or a sudden lack of memory can stall instructions. Equivalently, if Turbo Boost is turned on by the hardware, computation

[1]The output signals of the plant are usually the values of measurable goals, while the input signals of the plant can be the configuration of the plant itself – *e.g.*, a vector composed of elements like the algorithm used to solve a specific problem, the amount of resource allocated to a running application, the length of a queue, and much more.

might be accelerated. These quantities that act on the measured output but are not under the direct control of the controller, are called *disturbances* in control terms, and denoted by $d(k)$ in Figure 2. It is not necessary to identify them at design time, but it is important to acknowledge their existence. One of the purposes of control theory is to provably minimize the effect of the disturbances on the controlled output variables.

The similarities between self-adaptive systems and controlled systems, *i.e.* the coupling of a plant with its controller, are self-evident, with $\bar{y}(k)$ representing the system's goals and $u(k)$ the adaptation actions that will achieve these goals.

Merging the design of self-adaptive systems with the theory of controlling industrial plants can enhance the software engineering process with a variety of mathematically grounded adaptation laws. This has an impact on every stage of software development process, from requirements analysis to design, implementation and testing. At the requirement analysis phase, the needs of stakeholders should be matched to control goals. At the design phase, controllers should become a first-class element together with the introduction of analytical views capturing the intended dynamic behavior of the software system. During implementation and quality assurance, the presence of controllers introduces new challenges for verification and testing. In the following, we sketch a development process that takes into account control theory as a powerful framework for developing self-adaptive systems.

III. Control design process

This section discusses the design of a feedback control strategy for an existing software system. We provide details on the steps of the process that one should follow to develop a self-adaptive system with control-theoretical guarantees. The overall process is depicted in Figure 3.

The process starts by defining the system's goals, which are inputs of the control system. Values corresponding to the current state of the satisfaction of these goals need to be quantifiable and measurable so that they can serve as feedback to the controller. The goals are denoted by $\bar{y}(k)$ (which in general depends on the time step k), while the measurements of their satisfaction is denoted as $y(k)$ in Figure 2. The next step is identification of the software adaptation features (*i.e.* knobs or actuators) that can be changed at runtime, as denoted by $u(k)$ in Figure 2. The next step is to define a mathematical model that describes how changes in knob settings affect the measurable feedback. This model is denoted by P in Figure 2.

Given the system model, the next step is to synthesize a controller (C in Figure 2) with the most appropriate method among the many different ones that control theory offers.

The next step is to analyze the closed loop system and to prove that it behaves as desired. Informally, the closed loop system should reach and stabilize at its goal(s), and be robust to external variations and changes. These properties should be also verified in the presence of disturbances ($d(k)$ in Figure 2). If the desired properties are not exhibited, the process backtracks to one of the previous steps and a new iteration is started. Once the desired properties of the closed loop system are met, the controller should be implemented and tested, both in isolation and integrated with the system under control.

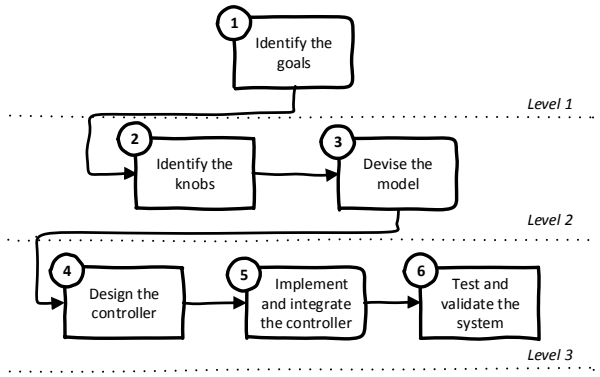

Fig. 3: Steps in the design and development of a control-based mechanism for self-adaptive systems. The different levels group steps into activities with tighter coupling.

Running Example: To illustrate the end-to-end control design process we will focus on the self-adaptation of a specific software system: (real-time) video encoding.

A video encoder processes a stream of video frames. Each frame is analyzed and the encoder chooses some frames as the encoding basis (I frames) and some frames to be encoded with respect to their differences with previous frames (P frames) or with both backward and forward frames (B frames). The encoding of I frames is very quick, since the data have to be copied directly into the resulting chunk of video, while encoding of P and B frames requires more operations. There are different encoding algorithms (*e.g.*, single pass, two passes, or three passes), and several parameters that can be tuned for each algorithm. For example, the resolution of the resulting frames can be tuned, along with the scene cut (a parameter that controls the insertion of I frames).

The video encoding example allows us to discuss different goals and different types of knobs. The encoding algorithm belongs to a non-ordered set of possible values. In contrast, the frame resolution belongs to a set of values ordered with respect to goal satisfaction. If the goal is accuracy, the higher the resolution, the more accurate the encoding. If the goal is frame latency, the lower the resolution, the lower the latency. The amount of computing capacity given to the application belongs to a continuous bounded set, between zero and the number of cores on the machine[2].

The video encoding example has already been studied in control literature [25, 38, 53, 54, 63]. It presents some interesting challenges, like the tradeoff between encoding compression and quality, energy minimization, and the dependence on input data (the video stream can be almost static; *e.g.*, a conference talk, where little effort has to be done to encode

[2]The set is continuous if we assume to use a scheduler such as SCHED_DEADLINE, now in mainline linux kernel, that allows to distribute fractions of the CPUs [47].

B and P frames, or very dynamic; *e.g.*, a sporting event). In the following sections, this example demonstrates the steps required to design a control strategy for a software system.

A. Identify the goals

The first step is to define the quantifiable and measurable goals the controller is to achieve.

We distinguish between *functional* and *non-functional* goals [31]. A functional goal concerns *what* functionalities the system exhibits, under a certain condition. A non-functional goal concerns *how good* the system has to perform, *i.e.* the quality it is expected to exhibit. A quality may target a specific functional requirement – *e.g.*, reliability of specific message passing – or may describe a global concern over the entire system – *e.g.*, only authorized users can access the database. Performance, reliability, and energy consumption are typical non-functional requirements that can be quantitatively expressed. For example, one may specify a system's performance requirements by introducing response time and throughput metrics. Not all non-functional requirements are easily quantifiable, however. Security and usability are instances of this category[3], although in special situations they may be measured.

Control strategies can be defined for the satisfaction of both functional and non-functional requirements. Though it is not a general rule, the former are usually managed by discrete-event controllers, capable, for example, of planning the composition of available components to introduce new functionality (*e.g.*, [18, 19]). The latter are often handled by means of equation-based controllers, tuning relevant knobs to change quantitative properties of the software system, like its timing properties or its energy consumption (e.g., [25, 38]).

(1) The simplest type of goal for an equation-based controller is a reference value to track. In this case, the control objective is to keep a measurable quantity (response time, energy consumption, occupied disk space) as close as possible to the given reference value, called a *setpoint*[4]. For example, the frame rate of a video streamer should be kept constantly at a standard frame rate of 24, 25 or 30 frames per second, depending on the country and the broadcasting infrastructure.

(2) A second category of goals is a variation of the classic setpoint-based goal, where the goal resides in a specific range of interest; *e.g.*, the average frame rate should be between 23 and 32 frames per second. Usually it is easy to transform these goals into equivalent setpoint-goals with confidence intervals.

Requirements elicitation plays an important role in defining the setpoints; it is in fact a systematic way to identify the critical requirements from the viewpoint of the stakeholders. Toward this end a new class of requirements, called Awareness Requirements (AwReqs) have been proposed. AwReqs constrain the success/failure of the system's critical requirements [65]. For example, for a web video streaming system a goal "Serve High Definition Video" could have as an AwReq "SuccessRate(80%)". This means that 80% of the time, high definition video should be delivered to the clients.

(3) A third broad category of goals concern the minimization (or maximization) of a measurable quantity of the system. For example, we may want to deliver our service with the lowest possible energy consumption. Depending on the specific quantity under control, certain optimization problems can be reduced to setpoint tracking. In particular, if the range of the measured property to optimize is convex and bounded, a setpoint tracker required to keep the measured property at its minimum possible value will drive the process as close as possible to the target, that is, toward the minimum possible value. Optimization problems, however, usually require more complex control strategies, especially when the optimization of one or more properties is subject to constraints on the values of others; *e.g.*, minimize the response time while keeping the availability above a certain threshold[5].

In our running example, a potential goal of this type is minimizing the video encoder's energy consumption. This goal can also be coupled with the timing properties of the software, which becomes an extra constraint. The precise formulation would be the minimization of the energy under the constraint that the frame rate is kept equal to a specific value. This formulation can generate conflicting goals. Speeding up the encoding process requires exploiting the hardware more and consuming in general more power. Reaching the timing goal can have a negative effect on energy consumption.

Special attention has to be paid when there are conflicting goals. Two simple types of conflict resolution strategies are prioritization and cost function definition. In the former, multiple goals are ranked according to their importance so that, whenever it is not be feasible to satisfy all of them at once, the controller will give precedence to the satisfaction of higher priority goals first[6]. Cost functions are another common means to specify the resolution of conflicting goals. In this case the utility function specifies all the suitable tradeoffs between conflicting goals as the Pareto front of an optimization problem. Optimal controllers can handle this type of specification guaranteeing an optimal resolution of conflicts.

A problem that is closely related to goal setting is the specification of graceful degradation strategies. Assuming a goal becomes temporarily infeasible, different reactions are possible. The simplest strategy may be to keep the system as close as possible to the target value. Setpoint tracking intrinsically provides this feature. However, the stability of the closed loop system for infeasible goals should be evaluated carefully. Another classic strategy for graceful degradation consists of disabling secondary functionality to free resources for achieving the primary goals, as done, *e.g.*, in [42] for cloud applications.

B. Identify the knobs

The second step is to find the knobs that change software behavior. In control theory, the knobs are generally determined

[3] An extended discussion on functional and non-functional requirements can be found in [31].

[4] Setpoint tracking is a very well studied problem in control theory [49].

[5] In this case, optimal control, discussed in [73] is more appropriate.

[6] Several control strategies support the achievement of prioritized goals, for example daisy chain control [49].

by the fundamental dynamics of the physical system under control. For example, an inverted pendulum has a single knob: the torque applied at the pivot point. If the pendulum is on a cart, the position of the pivot point can also be changed by moving the cart. In both cases there is a single knob, either the torque, or the position of the cart.

In some cases, software systems exhibit the same property and there are clear knobs that one can use to change the behavior of the system. In the case of the video encoder there are many potential knobs: the amount of computing resources given to the encoding applications, the encoding algorithm, the sensitivity of the encoding algorithm to changes in the frame[7] and many more. In contrast to physical systems, software systems are relatively easy to modify. As a consequence, there are cases in which knobs can be relatively easily added or removed. Both when using existing knobs and when devising new ones, the main issue is how to identify the best knobs to control the software system. In [64], an elicitation methodology is proposed, and the impact of each knob is evaluated with reference to the design goals.

Another important point when identifying the knobs is quantifying each knob's timescale. Some knobs, like powering up a new virtual machine, require time to bring about an effect. The control strategy should be aware of this time requirement and incorporate it into its model of the system's behavior.

In the video encoder case, the computing capacity given to the application can be considered a knob. In this case, the timescale of the visible changes is almost immediate and one expects the control action to have a measurable effect after the first second (when the average frame rate is updated).

C. Devise the model

The next step in the control design process is developing a model for the software system under control. In general, this model captures the relationship between the knobs identified in Section III-B and the goals identified in Section III-A, representing how a change in knob setting affects the goals.

In control theory, the model is analytical, so the interaction is formally described by mathematical relationships. This could mean that logical formulas are used (especially for discrete event control strategies) or that dynamical models are written (generally for continuous- or discrete-time-based controllers). Dynamical models are used to synthesize continuous-time or discrete-time-based strategies. A dynamical model can be either in state space form or expressed as an input-output relationship.

In the first case, one must choose the *state variables* that track the system's history and its evolution in time. Examples include the length of a queue, the percentage of frames already encoded, or the current encoding speed. Given the state variables' current values and the system's input values, one can formally determine the output variables' values [57]. The choice of the state variables is not unique since an infinite number of equivalent representations can be found [3].

[7]This parameter specifies the amount of change in the frame before a new *I* frame is produced and affects at least the encoding speed, accuracy, and energy consumption.

Once the state variables are identified, the relationship between the input and the state variables should be written, together with the relationship between the state and the output variables. For example, if the input variable of a queue is the number of incoming requests and the output variable is the average service time, the state variable is the number of enqueued requests. In principle, one could have many equations describing different inputs' effect on different state variables and the state variables' effect on the measurable outputs. These equations form the dynamical model used for the controller design. Depending on the equations' structure, there are different types of models: linear and nonlinear, switching, or parameter varying. Some examples follow ($x(k)$ representing the state of the system at time k).

$$x(k+1) = 3 \cdot x(k) \cdot k + u(k) \tag{1}$$

$$x(k+1) = -0.5 \cdot x(k) + u(k) \tag{2}$$

$$x(k+1) = 3 \cdot x(k)^2 + x(k) \cdot u(k) \tag{3}$$

$$x(k+1) = -0.5 \cdot x(k) + 3 \tag{4}$$

To be more precise about the properties of the system, if the equations in the system do not have an explicit dependence on time, the system is *time-invariant*. If the equations directly depend on functions of time, the system is *time-varying*. Among the equations above, (1) is time-varying, while (2), (3) and (4) are time-invariant. When the equation is linear with respect to the state x and the input u, the system model is *linear*. If the function is nonlinear with respect to x and u, the corresponding model is *nonlinear*. (1), (3) and (4) are nonlinear, while (2) is linear. Notice that (4) is *affine* and does not correspond to a linear system. Affine systems can always be expressed as linear systems, using a change of variables.

Equations can also contain parameters that will vary over time. If the equations are linear, the corresponding systems can be analyzed as Linear Parameter Varying (LPV), for example as done in [67].

In the second case, the system is expressed as a direct *input-output relationship*. Contrary to the state variables-based representation, this representation of the system is unique. State-of-the-art techniques in *system identification* often capture a system's input-output relationship directly from data, without using state variables. In the linear case, the input-output relationship can be expressed as a *transfer function* [3]. Transfer functions are very useful for control design, because they encode the input-output relationship with an algebraic relationship, that can be used to assess the system's satisfaction of desired properties. The same properties can be verified using state-space model, but that requires solving linear systems, which is harder than finding the roots of a polynomial (the standard way to assess the stability of a system specified with a transfer function).

In writing the model one should also consider potential disturbances acting on the system. In the video encoding example, there are inherently uncontrollable factors, such as operating system context switching and platform garbage collection.

Disturbances can be either modeled together with the system or their existence can be acknowledged. In the former case, a better control strategy can be designed; *e.g.*, by coupling a feedback controller with a feedforward control strategy that measures the disturbance and cancels it. In the latter case, the feedback control strategy should reject the disturbances. For example, by increasing the number of cores given to a video encoder while a garbage collector is running, the controller can meet the desired frame rate without modeling the effect of the garbage collector or knowing when it starts and ends. The feedback mechanism would take care of that.

For the given video encoding example, we can use the resources assigned to the application as an actuator [39, 53]. We denote this value as speedup $s(k)$. As the goal is the encoding rate $y(k)$, we can express the relationship between these two quantities as

$$y(k) = \frac{s(k-1)}{w(k-1)} + d(k-1) \tag{5}$$

where $w(\cdot)$ is the possibly time varying application *workload*; *i.e.* the (nominal) time between two subsequent frame encodings, and $d(\cdot)$ is an exogenous disturbance accounting for any non-nominal behavior. If the workload parameter is considered constant $w(\cdot) = w$, the model is linear. Otherwise, this is an instance of a Linear Parameter Varying (LPV) model.

If w is a constant in (5), the transfer function of the plant, from s to y is

$$P(z) = \frac{Y(z)}{S(z)} = \frac{1}{w \cdot z} \tag{6}$$

where $S(z)$ is the Z transform of the control signal (the speedup), and $Y(z)$ the Z transform of the encoding rate. The Z transform converts a dynamical model from the time domain (where time is represented with a sequence of k discrete instants) to the frequency domain (where z represents the frequency).

Although there is no direct correspondence of such control-theoretical models to software engineering models used for adaptation, some parallels can be drawn. In software engineering, a system at this phase is typically represented by its structure in an architectural model [30, 44]. Following a gray-box approach, only the details relevant for adaptation are modeled. For example, if we use the architectural model for performance analysis, we will typically annotate each component with its measured/predicted service time.

Architectural models are then either directly employed or transformed into analytical models and used in the adaptation logic of the system. The adaptation logic is typically captured in Event-Condition-Action rules, which model the connection between the state (captured in architectural models, deployment models, etc.) and the goals (captured in goal models, constraints, etc.). Such a connection can be drawn since different system configurations (architecture or parameter configuration) fulfill the goals to different degrees. The concrete adaptation is either determined at design time (though often in an ad-doc manner) or inferred at runtime based on knowledge about which configuration "best" fulfills the goals. The connections between the state and the goals, however, are often not formalized in an analytical model.

Markov models are often used in the analysis phase of an adaptation loop. They are a class of state-based models describing systems that exhibit probabilistic behavior. There are a number of Markov models. We categorize them by the type of available control variables. The described models are for a discrete-time case, but for each model there exists a continuous-time counterpart.

In cases of fully deterministic systems, we can use *discrete-time Markov chains (DTMC)*. DTMCs describe the probability of moving between system states and the model itself does not include any controllable actions. There are several methods employing DTMCs in the context of controlling software [10, 22, 24, 26, 27]. A typical scenario includes verifying a DTMC model of a system against a property. If a violation is identified, a reconfiguration of the system is triggered. When some of the actions in the system are controllable, we can use *Markov decision processes (MDPs)*. MDPs extend Markov chains by modeling controllable actions using non-determinism. The resolution of the non-determinism is then used as a controller of the system. Controller synthesis for MDPs is a well researched subject with a number of synthesis method for various types of properties [4, 7, 46, 60]. Systems with multiple players with conflicting objectives where players exhibit probabilistic behavior can be formalized using stochastic games. *Stochastic Games* can be seen as an extension of MDPs where the control over states is divided between several players. Typically, we are interested in the resolution of the non-determinism that allows certain player to achieve his/her objective despite possibly hostile actions of other players. For stochastic games, controller synthesis has been a subject of recent interest [5, 21] including development of a tool for generating controllers in practice [13].

D. Design the controller

Different techniques can be used to design a controller. These techniques differ in the amount of information required to set up the control strategy, in the design process itself, and in the guarantees they offer [6].

The technique requiring the least information is *synthetic design*, which combines pre-designed control blocks. It often relies on the experience of the control specialist who looks at data concerning the controlled system and decides which blocks are necessary. For example, given a noisy output signal, a filter block could reduce the noise. Synthetic design usually starts with a basic control block and adds more to the system as more experiments are performed. Although the information required to set up the control strategy is very low, the expertise necessary to effectively design and tune such systems is high and both controller design experience and domain-specific knowledge are required. Furthermore, the formal guarantees this technique offers are limited [6]. This is due to the empirical nature of the controller design, where trial and error is applied and elements are added and removed. The main obstacle to formal guarantees is the interaction between the added elements, which is hard to predict a priori.

The second technique is a variation of the first one. It is based on the selection of a controller structure and it is often called *parameter optimization*. The only difference is that the choice of the controller parameters is often based on optimization strategies or on analytical tuning methods [69].

The last technique is often referred as *analytical design*, and it is based on the solution of an analytical problem. The information requirements greatly increase as an analytical model of the controlled entity is required. Based on an equation-based model, controller synthesis selects a suitable equation to link the output variables to the control variables. Depending on which analytical problem is used (the optimization of some quantities, the tracking of a setpoint, the rejection of disturbances), different guarantees are enforced with respect to the controlled system. In some cases, this process can be automatized and analytical control synthesis can be used with domain-specific knowledge but without prior control expertise [25]. This generally imposes limitations on the models to be used and on the obtained guarantees. Also, not all the control problems can be formulated in such a way that the solution can be derived automatically.

In the example of video encoding, since we have the transfer function of the plant, we can use a common methodology, called *loop shaping*, that consists of selecting the closed loop system's transfer function and deriving the controller expression based on the selected closed loop function and the plant's transfer function (it is thus a case of analytical design). We have modeled the plant, obtaining the transfer function of Equation (6), also denoted with $P(z)$. We define the transfer function of the controller as $C(z)$ and we denote with $G(z)$ the transfer function of the system after closing the loop (the transfer function that expresses the relationship between the desired value of the goals and the current real measurements). With the control design, we impose the closed loop transfer function $G(z)$ to have the expression

$$G(z) = \frac{Y(z)}{\bar{Y}(z)} = \frac{C(z) \cdot P(z)}{1 + C(z) \cdot P(z)} = \frac{1-p}{z-p} \tag{7}$$

where $\frac{C(z) \cdot P(z)}{1+C(z) \cdot P(z)}$ is the generic expression of the loop of Figure 2, $\frac{1-p}{z-p}$ is the particular expression that we want the loop to assume, p is a generic parameter and $\bar{Y}(z)$ is the transfer function of the setpoint. The static gain of the closed loop transfer function (that is obtained by computing the function when $z = 1$) is 1, meaning that the value of the setpoint will be transferred to the output unchanged. Also, we choose the closed loop transfer function so that the parameter p is the root of the denominator. This parameter will be used later to prove properties on the closed loop system and enforce stability and robustness to disturbances.

Computing $C(z)$ from Equation (7), substituting Equation (6) to $P(z)$, one obtains

$$C(z) = w \frac{(1-p) \cdot z}{z-1}. \tag{8}$$

Moving from the transfer function domain back to the time domain, this corresponds to the following control law.

$$s(k) = s(k-1) + w \cdot (1-p) \left[\bar{y}(k) - y(k)\right]. \tag{9}$$

Equation (9) requires us to know the value of w, or to estimate it with other techniques. For now, we assume we know the correct value of the parameter.

E. Formally evaluating the closed loop system

From a software engineer's perspective, a controller should provide the following properties [17, 25, 27]:

Setpoint Tracking. The setpoint is a translation of the goals to be achieved. For example, the system can be considered responsive when its user-perceived latency is below one second. Here, the setpoint is the one second value for maximum user-perceived response time. In general, the self-adaptive system should achieve the specified setpoint, whenever this is possible. If the setpoint is changed during the software's lifetime, the controlled system should react to this change and achieve the new setpoint. Whenever the setpoint is not reachable the controller should make the measured value $y(k)$ is as close as possible to the desired value $\bar{y}(k)$.

Transient behavior. Control theory not only guarantees that the setpoint is reached but also *how* this happens. The system's behavior during the initialization phase or when an abrupt change happens is usually called the "transient of the response". For example, it is possible to enforce that the response of the system does not oscillate around the setpoint, but is always below (or above) it.

Robustness to inaccurate or delayed measurements. Oftentimes, in a real system, obtaining accurate and punctual measurements is very costly, for example because the system is split in several parts and information has to be aggregated to provide a reliable measurement of the system status. A controlled system's (in control terms a closed-loop system composed by a plant and its controller) ability to cope with inaccurate measurements or with late measurements is called robustness. The controller should behave correctly despite transient errors or delayed data.

Disturbance rejection. In control terms a disturbance is everything that affects the closed-loop system other than the controller's action. For example, when a virtual machine provider places a machine belonging to a different software application onto the same physical machine as the target software, the performance of the controlled software may change due to interference [32]. Disturbances should be properly rejected by the control system, in the sense that the control variable should avoid any effect of this external interference on the goal. In the video encoding example, the controller should be able to distinguish between a drastic change of scene (like the switch between athletes and commentators in a sport event) and a temporary slow down due to the switch between speakers in a conference recording. In the virtual machine example, the controller should be able to distinguish between a transient migration that is slowing down the software for a limited period of time and a persistent co-location that requires action to be taken to guarantee the goal satisfaction.

These high level objectives have counterparts in control terminology and their satisfaction can be mapped into the "by design" satisfaction of the following properties [17, 66]:

Stability. An equilibrium is asymptotically stable when it the system tends to reach it and remains there, irregardless of the initial conditions. This means that the system output converges to a specific value as time tends to infinity. This equilibrium point should ideally be the specified setpoint value.

Absence of overshooting. An overshoot occurs when the system exceeds the setpoint before convergence. Controllers can be designed to avoid overshooting whenever necessary. This could also avoid unnecessary costs (for example when the control variable is a certain number of virtual machines to be fired up for a specific software application).

Guaranteed settling time. Settling time refers to the time required for the system to reach the stable equilibrium. The settling time can be guaranteed to be lower than a specific value when the controller is designed.

Robustness. A robust control system converges to the setpoint despite the underlying model's imprecision. This is very important whenever disturbances have to be rejected and the system has to make decisions with inaccurate measurements.

These four properties can be analytically guaranteed, based on the mathematical definition of the control system and the software. A self-adaptive system designed with the aid of control theory should provide formal quantitative guarantees on its convergence, on the time to obtain the goal, and on its robustness in the face of errors and noise.

In the case of discrete-time control systems depicted in Figure 2, it is possible to analytically express the closed loop transfer function that relates the goal $\bar{y}(k)$ as input and the measured value $y(k)$ as output. This has the form of Equation (7) and, factoring the denominator, can be rewritten without loss of generality as

$$G(z) = \frac{f(z)}{\prod_{i\in[1,m]}(z - p_i)} \tag{10}$$

where $f(z)$ is a generic function of z and the denominator is the product of m factors $(z - p_i)$. The values of p_i are the *poles* of the closed loop system. The poles can be real or complex numbers. To guarantee stability for discrete-time control systems, the poles p_i should lay in the unit circle (the circle with radius 1 of the complex plane). If one restricts to the real number case, this means that to guarantee stability, the poles should lay in the open interval $(-1, 1)$. For a more detailed explanation the reader can refer to [3]. Analogous analytical formulations are available for all the mentioned properties, hence it is possible to check if the original requirements are satisfied.

In the example of video encoding, the closed loop transfer function has the expression of Equation (7),

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

that has one pole in $z = p$. Depending on the choice of p, the system can be proven to be stable or unstable. If $-1 < p < 1$, the system is stable, therefore it reaches the setpoint. Moreover, if the value of p satisfies $0 \leq p < 1$ there are no oscillations in the transient response.

The settling time of the controlled system is also determined from Equation (7). We can now convert the model from the frequency domain back to the time domain by appling the inverse Z transform. The inverse transform of the closed-loop transfer function $G(z)$ gives

$$y(k) = \bar{y} \cdot (1 - p^k) \tag{11}$$

As k increases the system approaches $\bar{y}$, as confirmed by the stability and setpoint tracking properties. We define the settling time as the time it takes the system to achieve $(100 - \epsilon)\%$ of the final value of $\bar{y}$, which means that 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 (11), the first value of k for which our output enters the ϵ confidence zone is

$$k_\epsilon = \frac{\log 0.01\epsilon}{\log |p|} \tag{12}$$

which means that after k_ϵ control steps the signal reaches the confidence zone. That value depends on ϵ, which is often 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 how fast the system will reach its equilibrium. Indeed, the pole's value p can 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.

Whenever the desired properties are not satisfied, one can step back in the design process and design a different controller, as discussed in Section III-D. In some other cases, the model can be refined or a more comprehensive model can be used, as discussed in Section III-C. The use of a more complex model can capture a part of the self-adaptive software system that can be necessary to provide formal guarantees on the time behavior of the system. Finally, in some cases, one can go back and add a different knob, as discussed in Section III-B, to have better control over the goals of the software system.

F. Implement and integrate the controller

The next step after the controller design is its implementation and integration with the system under control. Although this step appears to be quite straightforward, it has also been called "*the hard part*" [33].

One of the main problem is that the implementation team (software engineers) often works independently from the control team (control experts) [50]. The transition from control algorithms, which are typically in form of formulas or simulation results, into software is a non-trivial process. It involves many ad-hoc decisions, not only in the controller implementation itself (*e.g.*, types of state variables), but also in the implementation of the accompanying code that is responsible for the integration. This becomes particularly challenging in the case of remotely distributed systems.

Controller Implementation: Much of today's control code is not handcrafted, but automatically generated from block

diagrams and equations by tools that control engineers use in their daily routine. Among others, these tools include MATLAB/Simulink[8], Modelica[9], and SCADE[10]. Such a use of models as design blueprints differentiates model-*based* from model-*driven* development approaches. While both use models for design, communication, documentation, and analysis purposes, model-driven approaches also use them to automatically derive (parts of) the implementation via code generation.

Code generation is performed in both "classical" control (control of physical plants) and in control of software systems. As an example from classical control, MATLAB/Simulink can generate efficient Hardware Description Language (HDL), Programmable Logic Controller (PLC), or C/C++ code. In control of software systems, domain-specific languages (DSLs) have been proposed to model feedback loops. The resulting software models are used to generate code that can be traced back to the models [45] or to directly execute the feedback loop via model interpretation [70]. Another popular approach in software engineering is the use of frameworks (cf. [62]) for control-theoretical controllers, such as [50] that is based on Ptolemy 2[11], or for rule- and utility-based controllers such as [30]. Finally, there are also existing libraries for various programming languages that support implementing common controllers, such as for Python[12].

Regardless of the specific implementation technology and the degree of automation in translating the control algorithm to runnable code, there are some recurring issues in every controller implementation. Namely:

Actuator saturation occurs when the controlled system is unable to follow the controller's output. This happens when a control knob is fully engaged. In classical control, actuator saturation is a direct result of a physical constraint (*e.g.*, a valve can only open up to a certain point and not more); in software systems control, similar constraints can be observed as well (*e.g.*, the number of servers used cannot exceed the number of available servers in the cluster).

Integrator windup is a direct consequence of the actuator saturation and occurs in controllers with integral terms (the *I* term in a PID controller). During actuator saturation, the integral control accumulates significant error, which causes a delay in error tracking when the system under control recovers to the point that the actuator is no longer saturated. Therefore, a conditional integrator (or integrator clamping) should be used to avoid the windup.

Integrator preloading is similar to windup and can occur (i) during system initialization, or (ii) when there is a significant change in the set point. In both cases, the integral term should be preloaded with a value to smooth the setpoint change.

Apart from dealing with the common problems presented above, the controller implementation has to be robust. Common mechanisms to achieve robustness are (i) determination and disregard of invalid input signals (*e.g.*, out-of-bound values), (ii) preventing duplicate control actions, and (iii) graceful degradation in case design-time assumptions are invalidated (*e.g.*, when the actual computation and communication jitter is larger than assumed by controller designers).

Controller Integration: After a controller has been implemented, it has to be integrated with the rest of the system. This includes wiring all the responsible components together and ensuring that the measurements are consistently collected across all the sources and that adaptation actions are correctly coordinated.

Regarding the running example, in the simplest case, this involves (i) instrumenting the video encoding software with appropriate sensors (*e.g.*, to observe the frame rate) and actuators (*e.g.*, to adapt the encoding quality) [53], and (ii) providing a runtime with a protocol that connects these sensors and effectors with the controller.

When integrating a controller into a system, there are two basic approaches to follow, with respect to the separation of concerns between the controller and the system under control [62]: (i) intertwine the control logic with the system under control – *internal* control, and (ii) externalize the control logic into a "controller" component and use sensor and actuator probes to connect the controller and the system under control – *external* control.

The external control provides a clear separation of concerns between the application and adaptation logic. Its advantages with respect to maintainability, substitutability, and reuse of the adaptation engine and associated processes makes it the preferred engineering choice [62]. It also allows building adaptation on top of legacy systems where the source code might be unavailable. The main drawback of external control is the assumption that the target system can provide (or can be instrumented to provide) all the necessary endpoints for its observation and consequent modification. This assumption seems reasonable since many systems already provide some interfaces (*e.g.*, tools, services, APIs) for their observation and adjustment [30], or could be instrumented to provide them (*e.g.*, by using aspect-oriented approaches). There is also a potential performance penalty as a consequence of using external interface or running some extra components and connectors which cannot be tolerated in some resource-constrained environments (*e.g.*, in embedded devices where memory footprint and transmission delays matters).

The separation of concerns in the external approach also makes it possible to provide more systematic methods for control integration by leveraging model-driven engineering techniques and domain-specific modeling [45, 70]. Essentially, these solutions raise the level of abstraction on which the feedback control is described and therefore make it amenable to automated analysis as well as complete integration code synthesis.

G. Test and validate the system

The next step involves the test and validation of the controller. This can be divided into two broad categories. first, one needs to test the controller itself and check that it

[8]http://www.mathworks.fr/products/simulink/
[9]https://www.modelica.org/
[10]http://www.esterel-technologies.com/products/scade-suite/
[11]http://ptolemy.eecs.berkeley.edu/ptolemyII/
[12]http://sourceforge.net/projects/python-control/

does the correct thing. A part of this is already done with proving the properties of the closed loop system. This proof, however, does not test the controller implementation. Second, is the controller verification together with the system under control, to understand if the controller can deliver the promised properties. This should be true, in general, but there might have been model discrepancies that have been overlooked during the design process, therefore validation is needed despite the analytical guarantees given by control theory.

Verification and validation of the controller: Static analysis and verification techniques can be used to assess both the controller code's conformance to its intended behavior and the absence of numerical errors due to the specificity of different programming languages and execution architectures.

There is a growing area of verification theories and tools focusing on real analysis and differential equations. The most recent advancements include Satisfiability Modulo Theory (SMT), Ordinary Differential Equations (ODEs) and hybrid model checking. The former can be used to verify if a system described by means of a set of differential equations can reach certain desirable (or undesirable) states, within a set finite accuracy [29]. In terms of scalability, SMT approaches over ODEs have been proved to scale up to hundreds of differential equations. Hybrid model checking is instead focused on the verification of properties for hybrid systems [61], which can, in general, be defined as finite automata equipped with variables that evolve continuously over time according to dynamic laws. These formalisms are useful to match a system's different dynamic behaviors with its current configuration, and can be especially valuable to study and verify switching controllers and the co-existence of discrete-event and equation-based ones. Current hybrid model checkers are usually limited to linear differential equations [28, 36].

Controller code usually relies on numerical routines. Using the primitives of general purpose programming languages to develop complex numerical procedures introduces unavoidable source of uncertainties, including the common issues related to finite numerical precision of their implementations. Some tools (*e.g.*, jpf-numeric) use model checking techniques to identify possible sources of numerical problems for programs implemented with general-purpose languages (*e.g.*, Java) and produce counter examples helping the developer with reproducing and fixing problems. Moreover, modern SMT tools can be used at compile time to verify the occurrence of numerical problems and automatically provide fixes guaranteeing the final results of the procedures to be correct up to a target precision [15]. Ad-hoc languages for implementing numerical routines: apart from the code-generation functions and API provided by established mathematical programming environments, such as Matlab, Maple, or Mathematica, some extensions to general purpose programming languages provide constructs useful for specifying mathematical solutions that may support coding and compiler-level verification and optimization of controllers.

Verification and validation of the controlled software system: Once the controller itself is verified, it is necessary to verify the controller implementation together with the system under control. This can be done by means of extensive experiments but the process can be supported by different tools. Each tool comes with its own drawback and guarantees.

Rigorous analysis: one can use tools like the *scenario theory* [9, 11, 12, 56] to provide probabilistic guarantees on the behavior of the controlled system together with the control strategy. The performance evaluation can be formulated as a *chance constrained optimization problem* and an approximate solution can be obtained by means of scenario theory. If this approach is taken, the software system's performance can be guaranteed to be in specific bounds with a given probability. This, for example, allows for quantifying the probability that the proposed solution is fulfilling the Service Level Agreements in the case of failures or unexpected behaviors. This type of analysis requires performing an extreme number of experiments, varying many sources of randomness in the system, to cover potentially many cases with the randomized experiments. It is advisable to use this tool when one needs formal guarantees that the implementation meets specific requirements in all the possible conditions and when it is not too costly to experiment with the production-ready system.

Empirical study: other types of analysis are based on systematic testing. One common way to validate a controller implementation and its system under control is to show statistical evidence, for example, using cumulative distribution functions, as done in [42, 43]. In this work, less experiment effort is required are than for the rigorous analysis mentioned previously. Based on the results of these experiments, one can compute the empirical probability distributions of the goals. If the most important test cases are covered with the experiments, an empirical guarantee of the system behavior is obtained. This type of guarantee, however, does not offer any bound or any formal characterization of the system's behavior. In fact, if the system experiences different conditions than those tested, it might expose an undiscovered bug and misbehave.

IV. Conclusions

This paper discussed how control theory results can be integrated in the design of self-adaptive software. We have clearly defined the steps that are required to develop control strategies for a software system and used an example to clarify and discuss how each phase of the control loop design process can be cast in a practical application. We believe that this knowledge systematization is the first step towards introducing control strategies in the design of self-adaptive software systems that offer formal guarantees on their behavior.

Acknowledgements: This work was supported by: the Swedish Research Council (VR) with the projects "Cloud Control" and "Power and temperature control for large-scale computing infrastructures", the LCCC Linnaeus Center, the ELLIIT Excellence Center, the Datalyse project, the Irish Centre for Cloud Computing and Commerce (IC4), the European Union's Seventh Framework Programme FP7/2007-2013 under grant agreement 610802 "CloudWave" and the ERC advanced grants 267856 "Lucretius: Foundations for Software Evolution" and 246967 "VERIWARE".

REFERENCES

[1] T. Abdelzaher, Y. Diao, J. L. Hellerstein, C. Lu, and X. Zhu. "Introduction to Control Theory And Its Application to Computing Systems". In: *Performance Modeling and Engineering*. Ed. by Z. Liu and C. Xia. Springer US, 2008, pp. 185–215. DOI: 10.1007/978-0-387-79361-0_7.

[2] D. Arcelli, V. Cortellessa, A. Filieri, and A. Leva. "Control Theory for Model-based Performance-driven Software Adaptation". In: QoSA. ACM, 2015 - to appear.

[3] K. J. Åström and R. M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press, 2008. URL: http://resolver.caltech.edu/CaltechBOOK:2008.003.

[4] C. Baier, M. Grer, M. Leucker, B. Bollig, and F. Ciesinski. "Controller synthesis for probabilistic systems". In: *In Proceedings of IFIP TCS2004*. Kluwer, 2004. DOI: 10.1007/1-4020-8141-3_38.

[5] N. Basset, M. Kwiatkowska, and C. Wiltsche. "Compositional Controller Synthesis for Stochastic Games". In: *25th International Conference on Concurrency Theory (CONCUR'14)*. Vol. 8704. LNCS. Springer, 2014, pp. 173–187. DOI: 10.1007/978-3-662-44584-6_13.

[6] S. Boyd, C. Baratt, and S. Norman. "Linear controller design: limits of performance via convex optimization". In: *Proceedings of the IEEE* 78.3 (1990), pp. 529–574. DOI: 10.1109/5.52229.

[7] T. Brázdil, V. Forejt, and A. Kučera. "Controller Synthesis and Verification for Markov Decision Processes with Qualitative Branching Time Objectives". In: vol. 5126. LNCS. Springer, 2008, pp. 148–159. DOI: 10.1007/978-3-540-70583-3_13.

[8] Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. "Engineering Self-Adaptive Systems through Feedback Loops". In: *Software Engineering for Self-Adaptive Systems*. Vol. 5525. LNCS. Springer, 2009, pp. 48–70. DOI: 10.1007/978-3-642-02161-9_3.

[9] G. Calafiore and M. C. Campi. "Uncertain convex programs: randomized solutions and confidence levels". In: *Mathematical Programming* 102.1 (2005), pp. 25–46. DOI: 10.1007/s10107-003-0499-y.

[10] R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola. "Self-adaptive software needs quantitative verification at runtime". In: *Communications of the ACM* 55.9 (2012), pp. 69–77. DOI: 10.1145/2330667.2330686.

[11] M. C. Campi and S. Garatti. "A Sampling-and-Discarding Approach to Chance-Constrained Optimization: Feasibility and Optimality". In: *Journal of Optimization Theory and Applications* 148.2 (2011), pp. 257–280. DOI: 10.1007/s10957-010-9754-6.

[12] M. C. Campi, S. Garatti, and M. Prandini. "The scenario approach for systems and control design". In: *Annual Reviews in Control* 33.2 (2009), pp. 149–157. DOI: 10.1016/j.arcontrol.2009.07.001.

[13] T. Chen, V. Forejt, M. Kwiatkowska, D. Parker, and A. Simaitis. "PRISM-games: A Model Checker for Stochastic Multi-Player Games". In: ed. by N. Piterman and S. Smolka. Vol. 7795. TACAS. Springer, 2013, pp. 185–191. DOI: 10.1007/978-3-642-36742-7_13.

[14] B. H. Cheng et al. "Software Engineering for Self-Adaptive Systems". In: *Software Engineering for Self-Adaptive Systems*. Ed. by B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, and J. Magee. Springer-Verlag, 2009. Chap. Software Engineering for Self-Adaptive Systems: A Research Roadmap, pp. 1–26. DOI: 10.1007/978-3-642-02161-9_1.

[15] E. Darulova and V. Kuncak. "Sound Compilation of Reals". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. ACM, 2014, pp. 235–248. DOI: 10.1145/2535838.2535874.

[16] Y. Diao, J. Hellerstein, S. Parekh, R. Griffith, G. Kaiser, and D. Phung. "Self-managing systems: a control theory foundation". In: *ECBS Workshop*. 2005, pp. 441–448. DOI: 10.1109/ECBS.2005.60.

[17] Y. Diao, J. L. Hellerstein, S. Parekh, R. Griffith, G. E. Kaiser, and D. Phung. "A Control Theory Foundation for Self-managing Computing Systems". In: *IEEE J.Sel. A. Commun.* 23.12 (Sept. 2006), pp. 2213–2222. DOI: 10.1109/JSAC.2005.857206.

[18] N. D'Ippolito, V. A. Braberman, N. Piterman, and S. Uchitel. "Synthesis of live behaviour models for fallible domains". In: ICSE. ACM, 2011, pp. 211–220. DOI: 10.1145/1985793.1985823.

[19] N. R. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel. "Synthesis of Live Behaviour Models". In: FSE. ACM, 2010, pp. 77–86. DOI: 10.1145/1882291.1882305.

[20] R. Dorf and R. Bishop. *Modern control systems*. Prentice Hall, 2008.

[21] K. Draeger, V. Forejt, M. Kwiatkowska, D. Parker, and M. Ujma. "Permissive Controller Synthesis for Probabilistic Systems". In: vol. 8413. TACAS. Springer, 2014, pp. 531–546. DOI: 10.1007/978-3-642-54862-8_44.

[22] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. "Model evolution by run-time parameter adaptation". In: ICSE. IEEE. 2009, pp. 111–121. DOI: 10.1109/ICSE.2009.5070513.

[23] A. Filieri, C. Ghezzi, A. Leva, and M. Maggio. "Reliability-driven dynamic binding via feedback control". In: SEAMS. 2012, pp. 43–52. DOI: 10.1109/SEAMS.2012.6224390.

[24] A. Filieri, C. Ghezzi, and G. Tamburrelli. "A formal approach to adaptive software: continuous assurance of non-functional requirements". In: *Formal Asp. Comput.* 24.2 (2012), pp. 163–186. DOI: 10.1007/s00165-011-0207-2.

[25] A. Filieri, H. Hoffmann, and M. Maggio. "Automated Design of Self-adaptive Software with Control-theoretical Formal Guarantees". In: ICSE. ACM, 2014, pp. 299–310. DOI: 10.1145/2568225.2568272.

[26] A. Filieri, L. Grunske, and A. Leva. "Lightweight Adaptive Filtering for Efficient Learning and Updating of Probabilistic Models". In: ICSE. IEEE, 2015 - to appear.

[27] A. Filieri, C. Ghezzi, A. Leva, and M. Maggio. "Self-adaptive software meets control theory: A preliminary approach supporting reliability requirements". In: ASE. IEEE, 2011, pp. 283–292. DOI: 10.1109/ASE.2011.6100064.

[28] M. Franzle and C. Herde. "HySAT: An efficient proof engine for bounded model checking of hybrid systems". In: *Form Method Syst Des* 30.3 (2007), pp. 179–198. DOI: 10.1007/s10703-006-0031-0.

[29] S. Gao, S. Kong, and E. Clarke. "Satisfiability modulo ODEs". In: *Formal Methods in Computer-Aided Design (FMCAD), 2013*. 2013, pp. 105–112. DOI: 10.1109/FMCAD.2013.6679398.

[30] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. "Rainbow: architecture-based self-adaptation with reusable infrastructure". In: *Computer* 37.10 (2004), pp. 46–54. DOI: 10.1109/MC.2004.175.

[31] M. Glinz. "On Non-Functional Requirements". In: RE. 2007, pp. 21–26. DOI: 10.1109/RE.2007.45.

[32] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam. "Cuanta: Quantifying Effects of Shared On-chip Resource Interference for Consolidated Virtual Machines". In: SOCC. ACM, 2011, 22:1–22:14. DOI: 10.1145/2038916.2038938.

[33] J. L. Hellerstein. "Engineering Autonomic Systems". In: ICAC. ACM, 2009, pp. 75–76. DOI: 10.1145/1555228.1555254.

[34] J. L. Hellerstein, V. Morrison, and E. Eilebrecht. "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. DOI: 10.1145/1710115.1710123.

[35] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[36] T. Henzinger, P.-H. Ho, and H. Wong-Toi. "HyTech: A model checker for hybrid systems". In: vol. 1254. CAV. Springer, 1997, pp. 460–463. DOI: 10.1007/3-540-63166-6_48.

[37] H. Hoffmann et al. "Self-aware computing in the Angstrom processor". In: DAC. 2012, pp. 259–264. DOI: 10.1145/2228360.2228409.

[38] H. Hoffmann. "CoAdapt: Predictable Behavior for Accuracy-Aware Applications Running on Power-Aware Systems". In: ECRTS. 2014, pp. 223–232. DOI: 10.1109/ECRTS.2014.32.

[39] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal. "A generalized software framework for accurate and efficient management of performance goals". In: EMSOFT. 2013, pp. 1–10. DOI: 10.1109/EMSOFT.2013.6658597.

[40] 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. DOI: 10.1145/1380584.1380585.

[41] J. Kephart and D. Chess. “The vision of autonomic computing”. In: *Computer* 36.1 (2003), pp. 41–50. DOI: 10.1109/MC.2003.1160055.

[42] C. Klein, M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez. “Brownout: Building More Robust Cloud Applications”. In: ICSE. ACM, 2014, pp. 700–711. DOI: 10.1145/2568225.2568227.

[43] C. Klein, A. V. Papadopoulos, M. Dellkrantz, J. Durango, M. Maggio, K.-E. Arzen, F. Hernandez-Rodriguez, and E. Elmroth. “Improving Cloud Service Resilience Using Brownout-Aware Load-Balancing”. In: SRDS. IEEE, 2014, pp. 31–40. DOI: 10.1109/SRDS.2014.14.

[44] J. Kramer and J. Magee. “Self-Managed Systems: an Architectural Challenge”. In: FOSE. IEEE CS, 2007, pp. 259–268. DOI: 10.1109/FOSE.2007.19.

[45] F. Krikava, P. Collet, R. France, et al. “ACTRESS: Domain-Specific Modeling of Self-Adaptive Software Architectures”. In: *Symposium On Applied Computing, track on Dependable and Dependable and Adaptive Distributed Systems*. 2014.

[46] M. Kwiatkowska and D. Parker. “Automated Verification and Strategy Synthesis for Probabilistic Systems”. In: vol. 8172. ATVA. Springer, 2013, pp. 5–22. DOI: 10.1007/978-3-319-02444-8_2.

[47] J. Lelli, G. Lipari, D. Faggioli, and T. Cucinotta. “An efficient and scalable implementation of global EDF in Linux”. In: OSPERT. 2011.

[48] A. Leva, M. Maggio, A. V. Papadopoulos, and F. Terraneo. *Control-based operating system design*. Control Engineering Series. IET, 2013. DOI: 10.1049/PBCE089E.

[49] W. Levine. *The Control Systems Handbook, Second Edition: Control System Advanced Methods, Second Edition*. Electrical Engineering Handbook. Taylor and Francis, 2010.

[50] J Liu, J. Eker, and J. Janneck. “Actor-oriented control system design: A responsible framework perspective”. In: *IEEE Trans. Control Syst. Technol.* 12.2 (2004), pp. 250 –262. DOI: 10.1109/TCST.2004.824310.

[51] C. Lu, Y. Lu, T. Abdelzaher, J. Stankovic, and S. Son. “Feedback Control Architecture and Design Methodology for Service Delay Guarantees in Web Servers”. In: *IEEE Trans. Parallel Distrib. Syst.* 17.9 (2006), pp. 1014–1027.

[52] M. Maggio, H. Hoffmann, A. V. Papadopoulos, J. Panerati, M. D. Santambrogio, A. Agarwal, and A. Leva. “Comparison of Decision Making Strategies for Self-Optimization in Autonomic Computing Systems”. In: *ACM T Auton Adap Sys* 7.4 (2012), 36:1–36:32. DOI: 10.1145/2382570.2382572.

[53] M. Maggio, H. Hoffmann, M. D. Santambrogio, A. Agarwal, and A. Leva. “Power optimization in embedded systems via feedback control of resource allocation”. In: *IEEE Trans. Control Syst. Technol.* 21.1 (2013). DOI: 10.1109/TCST.2011.2177499.

[54] A. Muttreja, A. Raghunathan, S. Ravi, and N. K. Jha. “Automated Energy/Performance Macromodeling of Embedded Software”. In: *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* 26.3 (2007), pp. 542–552. DOI: 10.1109/TCAD.2006.883914.

[55] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. “An Architecture-Based Approach to Self-Adaptive Software”. In: *IEEE Intelligent Systems* 14.3 (1999), pp. 54–62. DOI: 10.1109/5254.769885.

[56] A. V. Papadopoulos and M. Prandini. “Model reduction of switched affine systems: a method based on balanced truncation and randomized optimization”. In: HSCC. ACM, 2014, pp. 113–122. DOI: 10.1145/2562059.2562131.

[57] A. V. Papadopoulos, M. Maggio, F. Terraneo, and A. Leva. “A Dynamic Modelling Framework for Control-based Computing System Design”. In: *Math Comp Model Dyn* (2014). DOI: 10.1080/13873954.2014.942785.

[58] S. Parekh. “Feedback Control Techniques for Performance Management of Computing Systems”. PhD thesis. 2010.

[59] T. Patikirikorala, A. Colman, J. Han, and L. Wang. “A Systematic Survey on the Design of Self-adaptive Software Systems Using Control Engineering Approaches”. In: SEAMS. IEEE, 2012, pp. 33–42. DOI: 10.1109/SEAMS.2012.6224389.

[60] M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. 1st. John Wiley & Sons, Inc., 1994.

[61] J.-F. Raskin. “An Introduction to Hybrid Automata”. In: *Handbook of Networked and Embedded Control Systems*. Control Engineering. Birkhauser Boston, 2005, pp. 491–517. DOI: 10.1007/0-8176-4404-0_21.

[62] M. Salehie and L. Tahvildari. “Self-adaptive Software: Landscape and Research Challenges”. In: *ACM Trans. Auton. Adapt. Syst.* 4.2 (May 2009), 14:1–14:42. DOI: 10.1145/1516533.1516538.

[63] M. Shafique, L. Bauer, and J. Henkel. “enBudget: A Run-Time Adaptive Predictive Energy-Budgeting scheme for energy-aware Motion Estimation in H.264/MPEG-4 AVC video encoder”. In: DATE. 2010, pp. 1725–1730. DOI: 10.1109/DATE.2010.5457093.

[64] V. E. S. Souza, A. Lapouchnian, and J. Mylopoulos. “System Identification for Adaptive Software Systems: a Requirements Engineering Perspective”. In: *Conceptual Modeling – ER 2011*. Vol. 6998. LNCS. Springer, 2011, pp. 346–361. DOI: 10.1007/978-3-642-24606-7_26.

[65] V. E. S. Souza, A. Lapouchnian, W. N. Robinson, and J. Mylopoulos. “Awareness Requirements for Adaptive Systems”. In: SEAMS. ACM, 2011, pp. 60–69. DOI: 10.1145/1988008.1988018.

[66] G. Stein. “Respect the unstable”. In: *Control Systems, IEEE* 23.4 (2003), pp. 12–25. DOI: 10.1109/MCS.2003.1213600.

[67] Q. Sun, G. Dai, and W. Pan. “LPV Model and Its Application in Web Server Performance Control”. In: *CSSE*. Vol. 3. IEEE CS, 2008, pp. 486–489. DOI: 10.1109/CSSE.2008.1219.

[68] M. Tanelli, D. Ardagna, and M. Lovera. “LPV model identification for Power Management of Web service Systems”. In: *MSC*. IEEE Control Systems Society, 2008, pp. 1171–1176. DOI: 10.1109/CCA.2008.4629616.

[69] A. Visioli. *Practical PID Control*. Advances in Industrial Control. Springer, 2006.

[70] T. Vogel and H. Giese. “Model-Driven Engineering of Self-Adaptive Software with EUREMA”. In: *ACM Trans. Auton. Adapt. Syst.* 8.4 (2014), 18:1–18:33. DOI: 10.1145/2555612.

[71] W.-P. Wang, D. Tipper, and S. Banerjee. “A simple approximation for modeling nonstationary queues”. In: vol. 1. INFOCOM. IEEE, 1996, pp. 255–262. DOI: 10.1109/INFCOM.1996.497901.

[72] D. Weyns and T. Ahmad. “Claims and Evidence for Architecture-based Self-adaptation: A Systematic Literature Review”. In: ECSA. Springer, 2013, pp. 249–265. DOI: 10.1007/978-3-642-39031-9_22.

[73] K. Zhou, J. C. Doyle, K. Glover, et al. *Robust and optimal control*. Vol. 40. Prentice Hall, 1996.

[74] X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, P. Padala, and K. Shin. “What does control theory bring to systems research?” In: *SIGOPS Oper. Syst. Rev.* 43 (1 2009), pp. 62–69. DOI: 10.1145/1496909.1496922.