Control Strategies for Self-Adaptive Software Systems

ANTONIO FILIERI, Imperial College London, UK
MARTINA MAGGIO, Lund University, SE
KONSTANTINOS ANGELOPOULOS, NICOLÁS D'IPPOLITO,
ILIAS GEROSTATHOPOULOS, ANDREAS BERNDT HEMPEL, HENRY HOFFMANN,
POOYAN JAMSHIDI, EVANGELIA KALYVIANAKI, CRISTIAN KLEIN, FILIP KRIKAVA,
SASA MISAILOVIC, ALESSANDRO V. PAPADOPOULOS, SUPRIO RAY,
AMIR M. SHARIFLOO, STEPAN SHEVTSOV, MATEUSZ UJMA, and THOMAS VOGEL

The pervasiveness and growing complexity of software systems are challenging software engineering to design systems that can adapt their behavior to withstand unpredictable, uncertain, and continuously changing execution environments. Control theoretical adaptation mechanisms have received growing interest from the software engineering community in the last few years for their mathematical grounding, allowing formal guarantees on the behavior of the controlled systems. However, most of these mechanisms are tailored to specific applications and can hardly be generalized into broadly applicable software design and development processes.

This article discusses a reference control design process, from goal identification to the verification and validation of the controlled system. A taxonomy of the main control strategies is introduced, analyzing their applicability to software adaptation for both functional and nonfunctional goals. A brief extract on how to deal with uncertainty complements the discussion. Finally, the article highlights a set of open challenges, both for the software engineering and the control theory research communities.

24

CCS Concepts: • **Software and its engineering → Software design engineering**; • **Computing methodologies → Modeling methodologies**; • Uncertainty quantification;

Additional Key Words and Phrases: Self-adaptive software, control theory, non-functional properties, formal methods

ACM Reference Format:
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 V. Papadopoulos, Suprio Ray, Amir M. Sharifloo, Stepan Shevtsov, Mateusz Ujma, and Thomas Vogel. 2017. Control strategies for self-adaptive software systems. ACM Trans. Auton. Adapt. Syst. 11, 4, Article 24 (February 2017), 31 pages.
DOI: http://dx.doi.org/10.1145/3024188

1. INTRODUCTION

The pervasiveness of software in every context of life is placing new challenges on software engineering. Highly dynamic environments, rapidly changing requirements, and unpredictable and uncertain operating conditions are pushing new paradigms for software design, leveraging runtime adaptation mechanisms to overcome the lack of knowledge at design time and design more robust software.

Authors' addresses: A. Filieri (corresponding author), Imperial College London, UK; email: a.filieri@imperial.ac.uk; M. Maggio (corresponding author), Lund University, SE; email: martina.maggio@control.lth.se.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2017 ACM 1556-4665/2017/02-ART24 $15.00
DOI: http://dx.doi.org/10.1145/3024188

The software engineering community has been working in the last decade on a multitude of approaches enhancing software systems with *self-adaptation* capabilities. However, most of them are focused on very specific problems and can hardly be generalized to broadly applicable methodologies [Filieri et al. 2014]. Furthermore, many proposed adaptation mechanisms lack the theoretical grounding needed to ensure their dependability besides the few specific cases they have been applied on Hellerstein et al. [2010].

Control theory has established in the last century a broad family of mathematically grounded techniques for designing controllers making industrial plants behave as expected. These controllers can provide formal guarantees about their effectiveness under precise assumptions on the operating conditions.

Although the similarities between the control of an industrial plant and the adaptation of a software system are self-evident, most of the attempts at applying off-the-shelf control theoretical results to software systems achieved very limited results, mostly tailored to specific applications and lacking rigorous assessment of the adequacy of the applied control strategies. The two main obstacles to achieve a "control theory for software systems" are (1) the difficulty in abstracting a software behavior in a mathematical form suitable for controller design and (2) the lack of methodologies in software engineering pursuing controllability as a first-class concern [Dorf and Bishop 2008; Diao et al. 2005; Zhu et al. 2009].

Analytical abstractions of software established for quality assurance often help to fill the gap between software models and dynamic models (i.e., differential or difference equations) [Wang et al. 1996; D'Ippolito et al. 2010]. However, software engineers often lack the mathematical background needed for a deeper understanding of these models. Goal formalization and knob identification are additional concerns of modeling, where specific characteristics have to be taken into account to achieve a suitable controllability of the system [Papadopoulos et al. 2015].

Several approaches have been trying to include control theoretical results into more general methodologies for designing adaptive software systems. The pioneering works in system engineering [Abdelzaher et al. 2008; Diao et al. 2006; Hellerstein et al. 2004] had the merit of spotlighting how control theoretical results can improve the design of computing systems. These contributions had a significant impact, especially on performance management and resource allocation. However, the new trends in self-adaptive software introduced new software models and a variety of quantitative and functional requirements beyond the scope of those works. More recently, methodological approaches for performance control [Parekh 2010] and the design of self-adaptive operating systems [Leva et al. 2013] have been proposed. Software engineering and autonomic computing have also highlighted the centrality of feedback loops for adaptive systems [Brun et al. 2009; Kephart and Chess 2003].

This article extends our previously published contribution [Filieri et al. 2015b], where we provided a *comprehensive analysis of the control theoretical design of self-adaptive software systems, matching the various phases of software development with the relevant background and techniques from control theory* (Section 2). The goal is to bootstrap the design of mathematically grounded controllers, sensors, and actuators for self-adaptive software systems to achieve formally provable effectiveness, efficiency, and robustness. More specifically, the aim of this article is to provide an overview of the control techniques that are well known in the control community and can be used or have been used for the design of self-adaptive software. In preparing this overview, we took the control perspective and summarized the result of many years of research in the control field.

To reach this aim, our previous work is here complemented by a detailed taxonomy of the most important control strategies (Section 3), discussing their applicability to specific software problems, and by a discussion of the related issue of modeling and formalizing uncertainty. In Section 4, we sketch some of the current contributions of

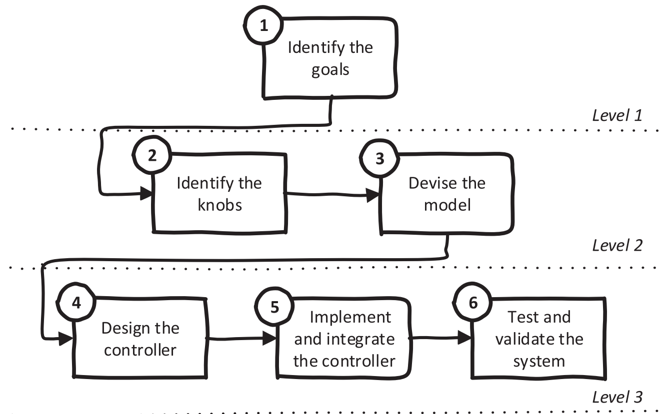

Fig. 1. Steps in the design and development of a control-based mechanism for self-adaptive systems. The different levels group steps into activities with tighter coupling. This figure originally appeared in Filieri et al. [2015b].

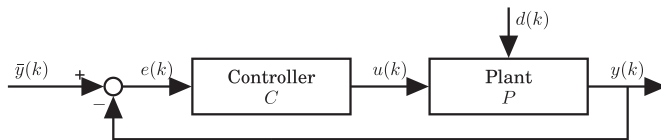

Fig. 2. Adaptive system: the *control perspective*. This figure originally appeared in Filieri et al. [2015b].

software engineering to controller design, implementation, and validation that could be valuable in bridging the gap between the two disciplines. Also, we identify a set of challenges for control of self-adaptive software (Section 5).

2. CONTROL DESIGN PROCESS

This section discusses the design of a control loop for an existing software system. The section covers and summarizes the control-based design process proposed in Filieri et al. [2015b], which the reader can refer to for additional details. The overall process is depicted in Figure 1, while Figure 2 shows the resulting control scheme. In our former contribution, we discussed more extensively some of the aspects of each step, providing also an example of the design process [Filieri et al. 2015b]. Here, on the contrary, we summarize the process and focus on a taxonomy of control strategies and on techniques to handle the inherent uncertainty present in software systems.

The process starts with the definition of the goals of the system that will be used by the controller as feedback signals and to measure and quantify when the adaptive system is fulfilling its objectives, denoted by $\bar{y}$ in Figure 2. The process continues with the identification of the quantities that can be changed at runtime to realize the adaptation features, denoted by u in Figure 2. These are usually called “knobs” or “actuators.”

Based on the identified goals and knobs, the next step is to define a model for the system, denoted by P in Figure 2. Given the model of the system, the next step is to synthesize a controller C 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 its behavior is the desired one. The closed-loop system should reach its goal(s) and be stable and sufficiently accurate and robust to variations and changes. These properties also should be verified in the presence of disturbances. 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.

2.1. Identify the Goals

The first question that should be answered is, what are the goals that one wants to achieve by building a control strategy on top of a software system? To build an effective control strategy, the goals have to be quantifiable and measurable.

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 setpoint.[1] A second category of goals is a variation of the classic setpoint-based goal, where the goal resides in a specific range of interest; for example, 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.

A third broad category of goals concerns 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 again 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. However, optimization problems 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).[2]

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 feasible to satisfy all of them at once, the controller will give precedence to the satisfaction of higher-priority goals first.[3] 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.

2.2. Identify the Knobs

The second step is to find the knobs that one can tweak in order to act toward the satisfaction of the specified goals. 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 Souza et al. [2011], 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 that can be used for control is the identification of each knob's timescale. Some of the knobs, like powering up a new virtual machine, require time to be effective. The control strategy should be aware of

[1] Setpoint tracking is a very well-studied problem in control theory [Levine 2010].
[2] In this case, optimal control, discussed in Section 3.2.2, is more appropriate.
[3] Several control strategies support the achievement of prioritized goals, for example, daisy chain control [Levine 2010].

that and base the action on prediction of the system's behavior or wait until the effect of the chosen action is measurable and quantifiable in the system.

2.3. Devise the Model

The next step in the control design process is the choice of a proper model for the software system under control. In general, to devise a control strategy, one would need a model of the relationship between the knobs identified in Section 2.2 and the goals identified in Section 2.1. Such a model represents the effect on the goals of a change in the knob values.

In control theory, the devised model is analytic, and therefore 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- 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 keep track of the past conditions in which the system was found and represent all the necessary knowledge about the system for understanding the evolution of its dynamics in time, for example, the length of a queue or the percentage of frames already encoded, or the current encoding speed. The knowledge about the state variables' current values and the input values of the system allows one to formally determine the values of the output variables [Papadopoulos et al. 2015]. The choice of the state variables may not be unique since an infinite number of equivalent representations can be found [Åström and Murray 2008]. 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 already enqueued requests. By knowing how many requests arrived since the last measurement and the service time of each request, one can determine the average service time of the request that was served in the last time interval analytically. The resulting equation is the system's model.

In principle, one could have many equations describing the effect of different inputs on different state variables and the effect of the state variables on the measurable outputs. These form a system of equations and constitute the dynamical model [Åström and Murray 2008] that is used for the controller design. Depending on the structure of this system, there are different types of models: linear and nonlinear [Åström 2008], switching, or parameter varying. 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) [Sun et al. 2008] ones or switching systems [Liberzon 2003]. In a switching system, there is a certain number of possible parameters, and while the system is evolving in time, the parameters can switch between the prescribed alternatives. In LPV systems, usually the alternatives are infinite and the parameters can take any possible value.

In the second case, the system is expressed as a direct *input-output relationship*. Contrary to the state-variable-based representation, this representation of the system is unique. State-of-the-art techniques in system identification often permit one to identify the input-output relationship of a system directly from data, without using state variables. In the linear case, the input-output relationship can be expressed as a transfer function [Åström and Murray 2008]. Transfer functions are very useful for control design, because they encode the input-output relationship with an algebraic

relationship, which in turn can be used to easily assess the satisfaction of desired properties of the system. The same properties can be assessed using state-space models, but they require 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, the possibility of disturbances acting on the system should also be considered. Disturbances can be modeled together with the system or their existence can be acknowledged. In the first case, a better control strategy can be designed, for example, coupling a feedback controller with a feed-forward control strategy that measures the disturbance and acts to cancel it. In the second case, the feedback control strategy should be able to reject some of the disturbances.

From a software engineering perspective, a system at this phase is typically represented by its structure in an architectural model (e.g., UML2, ADDL). Following a gray-box approach, only the details relevant for the specific adaptation concern are modeled. As an example, performance-based adaptation typically relies on performance prediction approaches for component-based architectures, which in turn rely on performance-annotated software architecture models [Balsamo et al. 2004; Brosig et al. 2015]. Architectural models are either directly employed or transformed into stochastic models (e.g., Queuing Networks [Tribastone 2013], Petri Nets [Ding et al. 2014], Markov models [Calinescu et al. 2012]) and used in the specification of the adaptation logic of the system.

A representative example of a modeling formalism used in software engineering, *Markov models* are a class of state-based models that can be generally used to describe systems that exhibit probabilistic behavior [Whittaker and Poore 1993]. There exist a number of Markov models. We categorize those models by the amount of control available. Although the models described later belong to the discrete-time case, for each model there also exists a continuous-time counterpart.

In cases of fully deterministic systems, we can use discrete-time Markov chains (DTMCs). DTMCs describe the probability of moving between system states and the model itself does not include any controllable actions. There exist several methods employing DTMCs in the context of controlling software [Calinescu et al. 2012; Epifani et al. 2009]. A typical scenario includes verifying a DTMC model of a system against a property. If a violation of the property has been 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 nondeterminism. The resolution of the nondeterminism is then used as a controller of the system. Controller synthesis for MDPs is a well-researched subject with a number of synthesis methods for various types of properties [Puterman 1994; Baier et al. 2004; Brázdil et al. 2008; Kwiatkowska and Parker 2013]. 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 nondeterminism that allows a certain player to achieve its objective despite possibly hostile actions of other players.

2.4. Design the Controller

There are different techniques that can be used to design a controller (in control engineering terms: to do control synthesis). These techniques differ in the amount of information required to set up the control strategy, in the design process itself, and in the guarantees that they can offer [Åström and Murray 2008].

The technique that requires the least amount of information is called *synthetic design*. Synthetic design consists of taking predesigned control blocks and combining them together. It often relies on the experience of the control specialist, which looks at experiments performed on the system to be controlled and decides which blocks are necessary. For example, when the output signal is very noisy, a block to be added could be a filter to reduce the noise and capture the original signal. 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. Despite not requiring much information, the formal guarantees that this technique offers are limited [Boyd et al. 1990]. 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 [Åström and Murray 2008].

The last technique is often referred as *analytical design*, and it is based on the solution of an analytical problem. The amount of necessary information greatly increases, since a model of the controlled entity is required. Based on the equation-based model, the 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 sometimes be used with domain-specific knowledge but without prior control expertise [Filieri et al. 2014]. This generally imposes limitations on the models to be used and on the obtained guarantees, and not all the control problems can be formulated in such a way that the solution can be derived automatically.

In control theory, controllers can be divided into two main classes: time-based and event-based controllers. The class of time-based controllers has been studied for many years and can be further divided into continuous-time and discrete-time controllers. Given the nature of software systems, discrete-time controllers—where signals are not continuous, but discrete—seem to be a better fit for control of software systems [Diao et al. 2005]. Nonetheless, there has been some effort in using continuous time models to represent and control software entities [Tribastone 2013]. In this class, the controller is an equation-based system that acts at prespecified time instants. When these time instants are not prespecified but depend on external inputs from the system, the controller belongs to the second class and it is an event-based controller. To complement these two classes, rule-based systems are often considered as a third, separate type of controllers. In control terms, rule-based—or knowledge-based—systems are a different paradigm that shares little with "classical" control. However, they have often been exploited to build self-adaptive software, and therefore we decided to include them in our control taxonomy.

2.5. Prove Properties of the Closed-Loop System

The properties provided by control-theoretical-designed adaptation strategies have been analyzed both from the software engineer perspective [Villegas et al. 2011; Filieri et al. 2014] and from the control engineer perspective [Diao et al. 2006; Parekh

et al. 2002]. Generally speaking, a control-based adaptation strategy would provide the properties it is designed for. For example, a control system that acts on the length of a key for an encryption algorithm is able to provide a different level of security. Taking the control perspective here, we focus on the translation of *control* properties into the corresponding software engineering ones. A control system usually pursues four main objectives:

—*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 1 second. The setpoint here is the value of 1 second for the maximum user-perceived response time. In general, the self-adaptive system should be able to achieve the specified setpoint whenever this setpoint is reachable. If the setpoint is changed during the lifetime of the software, the controlled system should react to this change and make sure that the new setpoint is reached. Whenever the setpoint is not reachable, the controller should make sure that the measured value $y(k)$ is as close as possible to the desired value $\bar{y}(k)$.
—*Transient behavior.* Control theory guarantee not only that the setpoint is reached but also *how* this happens. The behavior of the system 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. The ability of a controlled system (in control terms a closed-loop system composed by a plant and its controller) to cope with nonaccurate measurements or with data that is delayed in time is called robustness. The controller should behave correctly even when transient errors or delayed data is provided to it.
—*Disturbance rejection.* In control terms, a disturbance is everything that affects the closed-loop system other than the action of the controller. 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 [Govindan et al. 2011]. Disturbances should be properly rejected by the control system, in the sense that the control variable should be correctly chosen to 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 slowdown 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 colocation 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 [Diao et al. 2006; Stein 2003; Villegas et al. 2011]:

—*Stability.* A system is asymptotically stable when it tends to reach an equilibrium point, regardless 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 (e.g., 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 being imprecise. 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 of 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, these properties can be proved analytically. For more details, the reader is referred to Filieri et al. [2015b], which treats the matter with a software engineering example, and Åström and Murray [2008] for the control treatise.

Whenever the desired properties are not satisfied, one can step back in the design process and design a different controller, as discussed in Section 2.4. In some other cases, the model can be refined or a more comprehensive model can be used, as discussed in Section 2.3. 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 2.2, to have better control over the goals of the software system.

2.6. 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” [Hellerstein 2009].

One of the main problems is that the implementation team (software engineers) often works independently from the control team (control experts) [Liu et al. 2004]. The transition from control algorithms, which is typically in the form of formulas or simulation results, into software is a nontrivial 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.

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 attempt. Namely:

—*Actuator saturation* occurs when the system under control is unable to follow the output from the controller. 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 by the system under control 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 integral windup and can occur (1) during system initialization or (2) when there is a significant change in the setpoint of the controller. In both cases, the integral term should be preloaded with a value that would smooth the setpoint change.

Apart from dealing with the common problems presented previously, the controller implementation has to be robust. Common mechanisms to achieve robustness are (1) determination and disregard of invalid input signals (e.g., out-of-bound values), (2) account for actuator delays by preventing duplicate control actions, and (3) graceful degradation in case of invalidation of design-time assumptions (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 sensor data are consistently collected across all the sources and that adaptation actions are correctly coordinated.

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 [Salehie and Tahvildari 2009]: (1) intertwine the control logic with the system under control—*internal* control, and (2) externalize the control logic into a "controller" component and use sensor and actuator probes (if available) 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 make it the preferred engineering choice [Salehie and Tahvildari 2009]. It also allows building adaptation on top of legacy systems where the source code might be unavailable. The main drawback of external control is, however, in assuming 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 [Garlan et al. 2004b] 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 an external interface or running some extra components and connectors that cannot be tolerated in some resource-constrained environments (e.g., in embedded devices where memory footprint and transmission delays matter).

The separation of concerns in the external approach also makes it possible to provide more systematic ways for control integration by leveraging model-driven engineering techniques and domain-specific modeling [Vogel and Giese 2014; Krikava et al. 2014]. 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.

2.7. Test and Validate the System

The next step involves the testing and validation of the controller. This can be divided into two broad categories. On one hand, one needs to test the controller itself and check that it does the correct thing. A part of this is already done with proving the properties

of closed-loop systems. However, this does not test the controller implementation. The second part is the verification of the controller 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.

Static analysis and verification techniques can be used to assess both the conformance of the controller code to its intended behavior and the absence of numerical errors due to the specificity of different programming languages and execution architectures. For example, 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 achieve a target accuracy [Darulova and Kuncak 2014].

Also, there is a growing area of verification theories and tools focusing on real analysis and differential equations. The most recent advancements include satisfiability modulo 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 [Gao et al. 2013]. 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, which can in general be defined as finite automata equipped with variables that evolve continuously according to dynamical laws over time. These formalisms are, for example, useful to match different dynamical behaviors of a system with its current configuration, and can be valuable especially to study and verify switching controllers and the coexistence of discrete-event and equation-based ones. Current hybrid model checkers are usually limited to linear differential equations [Henzinger et al. 1997; Franzle and Herde 2007].

Once the controller is verified by itself, it is necessary to verify the controller implementation together with the system under control. This can be done by means of extensive experiments. Some 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 Klein et al. [2014]. In this and similar work, some experiments are conducted, in a lower number with respect to the rigorous analysis previously mentioned. Based on the results of these experiments, one can compute the empirical probability distributions of the goals.

It is also possible to use tools like the scenario theory [Campi and Garatti 2011; Papadopoulos et al. 2016] 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 the scenario theory. If this approach is taken, the performance of the software system can be guaranteed to be in specific bounds with a given probability.

3. CONTROLLER TAXONOMY

This section discusses a selection of different control strategies and concepts that can be applied for the solution of software adaptation problems. While a comprehensive survey of all the techniques established by control theory is beyond the scope of this paper, a discussion of the principal classes and dimensions of control solutions is provided as a reference for casting common software adaptation problems in the framework and terminology of control. We first start with a discussion on the control loop types in Section 3.1. We then delve into the implementation of different types of controllers.

Section 3.2 discusses the time-based controllers, Section 3.3 the event-based ones, and Section 3.4 the knowledge-based ones. Finally, Section 3.5 discusses uncertainty management strategies that can complement the mentioned techniques.

3.1. Loop Type

Loop-based controllers are defined by the way in which they incorporate feedback and system measurement. We identify four main types:

—*Open-loop control:* The control signal is based on a mathematical model, without any measurement. This scheme is used *only* with perfect knowledge of the system and outside environment—it cannot react to unforeseen disturbances.
—*Feed-forward control:* This approach is similar to open loop but includes measurements of disturbances. The controller rejects these disturbances' effects on the system. Perfect knowledge of the system model is assumed, and hence, unmeasurable disturbances cannot be rejected.
—*Feedback control:* This is the most-used structure for the control loop. The control input is computed as a function of the difference between measured system behavior and desired behavior. This scheme is usually able to handle unforeseen disturbances and uncertainty in the system. In addition to rejecting unmeasured disturbances, feedback control can also significantly change the behavior of the controlled system (e.g., stabilizing an (open-loop) unstable system).
—*Feedback and feed-forward control:* The control is computed as in the feedback case, but disturbance measurement is also incorporated to actively reduce disturbances' effects. The feedback and feed-forward controllers must be designed together.

More advanced control schemes are discussed in the literature [Scattolini 2009] and have been used to control computing systems [Patikirikorala et al. 2012].

3.2. Time-Based Control Techniques

This section reviews the most important time-based control techniques, providing hints on when a particular technique should be used. The focus is mainly on discrete-time control techniques, since these are the most suited for being applied for designing self-adaptive software systems.

3.2.1. Classical Feedback Control. One of the simplest controllers is the *bang-bang controller* (or on-off controller) [Artstein 1980]. The control signal can assume only two values: $\{-1, 1\}$ or {on,off}. Due to its simplicity, it was widely studied especially in optimal control problems [Hermes and Lasalle 1969].

Pole placement [Wittenmark et al. 2002] is a special case of state-feedback control that assumes all system states are measured and used to compute the control signal. Since the overall state vector cannot always be measured, this technique is typically combined with measurements used to reconstruct the states' values. To use this technique, a state-space system model must be available and some modifications are required in order to account for possible disturbances and model uncertainties.

A special case of pole placement design is so-called *deadbeat control* [Wittenmark et al. 2002]. This strategy drives the controlled output to the desired value in at most n time units, where n is the order of the controlled process.

The most common controller, however, is the *PID controller* [Åström and Hägglund 2006]. This type of controller covers about 90% [Wittenmark et al. 2002] of the industrial applications, due to its simplicity and flexibility. It is based on a very simple principle, which is to compute the control input $u(k)$ (see Figure 2) according to

the law

$$u(k) = u(k-1) + \underbrace{K\Delta e(k)}_{\text{proportional}} + \underbrace{\frac{Kh}{T_i}e(k)}_{\text{integral}} + \underbrace{\frac{KT_d}{h}\left[\Delta e(k) - \Delta e(k-1)\right]}_{\text{derivative}}, \tag{1}$$

where $e(k) = \bar{y}(k) - y(k)$ is the error between the desired and the actual behavior of the system, $\Delta e(k) = e(k) - e(k-1)$ is the variation of the error, h is the sampling time of the controller, and K, T_i, and T_d are the three parameters of the PID controller. Note that Equation (1) is a PID control law in velocity form (cf. Åström and Hägglund [2006]). The three terms in the control law of Equation (1) serve different purposes:.

—A higher proportional gain K generally increases the response time of the closed-loop system but makes it unstable.
—The integral term can mitigate steady-state deviations from the setpoint that a purely proportional controller cannot handle but can introduce overshoot in the system response.
—A derivative term improves response time and stability but may amplify the influence of measurement noise.

PID controllers are popular as they do not require an explicit system model but can be tuned on the basis of experiments and established heuristics [Åström and Hägglund 2006]. On the other hand, PID control does not generalize easily to MIMO systems.

An alternative technique is *loop shaping*. The control designer decides the closed-loop transfer function's shape and designs the controller accordingly. The desired closed-loop transfer function is typically related to some time-domain specifications, for example, the required settling time or the maximum disturbance amplification.

3.2.2. Optimal Control. In optimal control, the control value is obtained so as to minimize a cost function, possibly subject to some constraints. Typically, the objective is to maximize control performance, given prescribed guarantees [Morari and Zafiriou 1989; Zhou et al. 1996]. Whenever the cost function is a quadratic function and the constraints contain linear first-order dynamic constraints, the problem can be classified as a *Linear Quadratic (LQ)* optimal control problem. A special case is the *Linear Quadratic Regulator (LQR)* [Skogestad and Postlethwaite 2007].

A particularly successful heuristic for optimal control under constraints is *Model Predictive Control (MPC)* [Maciejowski 2002; Camacho and Alba 2013]. MPC predicts the future behavior from the current system state under a particular control action and selects the input sequence that minimizes the chosen cost function. Only the first step of that input sequence is applied and at the next time step the new system state is determined and the process repeated. This control strategy is also called *Receding Horizon Control (RHC)*.

Such techniques are widely used in industry, especially when physical constraints must be enforced. A good system model is required for MPC. Computational resources can be an issue, as solving the optimization problem for larger (or nonlinear) systems can take significant time. Variants exist that deal with disturbances and model uncertainties in different ways [Bemporad and Morari 1999; Goodwin et al. 2014].

Another class of optimal controllers is the $\mathcal{H}_\infty$ *control* [Skogestad and Postlethwaite 2007]. This class is important in many applications in which guarantees on the obtainable performance are crucial. While mature tools exist that can automatically synthesize $\mathcal{H}_\infty$ controllers, mathematical expertise is required to understand the advantages and limitations of $\mathcal{H}_\infty$ control.

3.2.3. Adaptive Control. *Adaptive control* is a class of techniques that allows a controller to adapt its behavior to time-varying or uncertain structural properties [Åström and Wittenmark 2013]. Adaptive controllers modify the control law at runtime to adapt to the discrepancies between the expected and the actual system behavior. Usually, adaptation mechanisms are built on top of existing controllers and modify the controller's parameters or select the most effective controller to form a set of possibly structurally different ones, depending on the detected operation point [Åström and Wittenmark 2013]. The control task is, however, still performed by the controller. Most of the approaches using adaptive control for self-adaptive software perform parametric adaptation; that is, the structure of the controller is not changed, only the values of its parameters according to an adaptation policy.

An adaptive controller combines an online parameter or system state estimator with a control law designed from the prior knowledge about the system. The way the estimates are combined with the control law gives rise to different approaches. *Gain scheduling* estimates the system's current operating region and then changes the controller's parameters based on that estimate. *Model Identification Adaptive Controllers (MIACs)* use online identification techniques to estimate a good system model and modify the controller accordingly. *Model Reference Adaptive Controllers (MRACs)* change the controller's parameters according to the discrepancy between the expected and actual system behavior with respect to a reference model.

The term "adaptive control" usually refers to a specific class of adjustment mechanisms enabling controllers to face unexpected behavior. This is very different, and usually more complex with respect to many "adaptation" policies that are used in software systems. Unfortunately, the two terms are very close but distinct, making communication between the two communities difficult.

3.2.4. More Complex Techniques. Most robust control methods typically deal with uncertainties in a "worst-case" sense, providing strong guarantees under fairly strong assumptions. *Stochastic control*, on the other hand, tries to minimize the influence of uncertainties, providing probabilistic guarantees on the satisfaction of its requirements (e.g., the system will satisfy achieve its goals with a probability larger than a certain threshold) [Kumar and Varaiya 1986]. The benefit is that assumptions on the uncertainties are generally much weaker; for example, instead of assuming bounded disturbance signals, only an assumption on the probability distribution is necessary.

The control techniques outlined in the previous section have mostly been developed for linear systems because of the complete mathematical theory existing for this model class. Since most real systems are nonlinear, *nonlinear control* is an active field in research and application [Khalil 2015]. Since nonlinearities can take so many different forms, nonlinear control methods are often tailored to specific model structures. Many of the methods designed for linear systems have also been applied to nonlinear systems, for example, nonlinear MPC [Grüne and Pannek 2011]. PID controllers in particular are often applied to nonlinear systems because of their easy tuning and good robustness properties. Additionally, many nonlinear systems only operate under certain conditions, often allowing one to linearize the nonlinear dynamics around such an *operating point* and working with the resulting linear model.

A particular class of nonlinear models that may be especially relevant in the context of software systems is so-called *hybrid systems* [Goebel et al. 2012; Labinaz et al. 1997]. They *combine* continuous state dynamics on a continuous- or discrete-time scale (e.g., number of requests in a queue) with intermittent discrete events (e.g., availability of more computing resources). Analysis and control of hybrid systems have to reconcile the continuous state dynamics with abrupt discrete changes in the system's state

or behavior. Despite hybrid systems gaining growing interest for modeling software behaviors [Zhan et al. 2013], verifying the properties of (controlled) hybrid systems is particularly challenging [Alur 2011], with restrictive decidability results for the more general classes of hybrid models [Henzinger et al. 1998]. A special case of a hybrid controller is the so-called *switching controller* [Liberzon 2003]. The idea of switching control is similar to adaptive control, but instead of modifying the parameters of the controllers, a switching policy decides how to switch between controllers of different structures. In software engineering terms, it is a structural adaptation that changes the control component itself. For a broader overview of the current state of the art on hybrid control see, for example, Lunze and Lamnabhi-Lagarrigue [2009].

3.3. Event-Based Controllers

Event-based controllers react to specific events by making decisions about the current system. This class includes controllers entirely designed in the event space, for example, logic controllers, and controllers that are event-based extensions of a continuous- or discrete-time control. Here, we briefly analyze this last class and then discuss "pure event" strategies.

In the control domain, there is a distinction between *event-triggered* and *self-triggered* control strategies [Heemels et al. 2012; Åström 2008; Lunze and Lehmann 2010]. An initial effort was devoted to studying what happens when a controller designed to be executed periodically is instead called when a certain event happens, for example, when a performance drop is experienced. More recently, the control community has put some effort into the systematic design of event-based implementations of feedback control laws [Tabuada 2007; Heemels et al. 2008; Leva and Papadopoulos 2013]. Event-triggered controllers are based on constant monitoring of some triggering conditions that could lead to the execution of the controller code. The controller can be designed with any of the mentioned strategies. On the contrary, self-triggered controllers are not based on constant monitor, but the controller, before terminating its executions, programs a new wakeup time [Camacho et al. 2010; Leva and Papadopoulos 2013]. Whenever the monitor is extremely expensive or the verification of the triggering condition is nontrivial, self-triggered control strategies can offer a better alternative to event-based ones. However, it must be noted that the design of the controller is often conducted with the same techniques used for continuous- and discrete-time control strategies (see Section 3.2).

In the computer science domain, the problem of event-based controller synthesis has been defined [Ramadge and Wonham 1989; Pnueli and Rosner 1989]. Event-based controller synthesis can be abstractly defined as follows: given a model of the assumed behavior of the environment (E) and a system goal (G), controller synthesis produces an operational behavior model for a component M that when executing in an environment consistent with the assumptions results in a system that is guaranteed to satisfy the goal—that is, $E \| M \models G$.

The main motivation for event-based controllers in self-adaptive systems is the need for high-level adaptation strategies and decision-level behavior plans. Thus, event-based controllers, in contrast to discrete-time ones, entail complex strategies to achieve high-level complex goals (e.g., architectural adaptation) that require a nontrivial combination of actions. Further, event-based controllers provide formal guarantees that ensure goals are satisfied.

To apply event-based controller synthesis requires having formal specification of the environment assumptions and system goals. Temporal logics (e.g., LTL [Pnueli 1977]) are a widely adopted formalism to specify environment assumptions and system goals in computer science, and more specifically in the software engineering community.

Temporal logics not only provide a framework that formalizes the entailment ($\models$) operator, central to the controller synthesis problem, but also allow for declarative and precise descriptions of high-level complex goals. Many have addressed the problem of synthesizing a controller for safety goals (e.g., Ramadge and Wonham [1989]), while others have introduced approaches for general LTL formulas (e.g., Pnueli and Rosner [1989]). Here we report on a number of approaches that incorporate (and thus guarantee) different subsets of LTL as system goals.

A number of architectural approaches for self-adaptive systems [Kramer and Magee 2007; Garlan et al. 2004a; Dashofy et al. 2002; Batista et al. 2005; Oreizy et al. 1999; Kang and Garlan 2014; Inverardi and Tivoli 2003; Braberman et al. 2015] have been proposed. At the heart of many such adaptation techniques, there is a component capable of designing at runtime a strategy for adapting to the changes in the environment, system, and requirements (e.g., Braberman et al. [2015]). Interestingly, no mechanisms for generating adaptation strategies is prescribed. In fact, depending on the knowledge on the environment and the type of goal, a wide range of event-based controller synthesis techniques could be applied.

There are a large variety of controller synthesis techniques [Pnueli and Rosner 1989] that guarantee the satisfaction of safety and even liveness [Piterman et al. 2006; D'ippolito et al. 2013] requirements within the constraints enforced by the problem domain, within the capabilities offered by the system, and under fairness and progress assumptions on the controller's environment.

Traditional techniques for controller synthesis are Boolean in the sense that a controller satisfies a set of goals or it does not. This two-valued view is limiting as, in general, there are multiple ways to satisfy a set of goals, leading to several levels of satisfaction with respect to certain preference notions—that is, quality attributes. Typically, such quality attributes are modeled by introducing a quantitative aspect to the system specification, imposing a preference order on the controllers that satisfy the qualitative part of the specification (e.g., Chatterjee et al. [2015]). Thus, the event-based synthesis procedure has both qualitative and quantitative aspects, the former modeling goals the controller must satisfy and the latter modeling the properties of the dynamical behavior exhibited by the system that are to be, for example, maximized.

In some cases, the environment behavior is stochastic. Event-based controller synthesis techniques for such models are normally based on the notion of stochastic games [Chatterjee et al. 2004]. The problem of stochastic synthesis has qualitative and quantitative components. The qualitative component is to answer if a controller can guarantee the satisfaction of a goal or if it has a positive probability of achieving the goal [Chatterjee et al. 2003]. The quantitative component, on the other hand, is to answer the exact value of the probability a controller has to fulfill the goals [Chatterjee et al. 2015]. Stochastic control problems have various forms depending on the type of environment the controller must interact with. When the controller is meant to interact with a purely stochastic environment, $1^{1/2}$ player games are used. Such games can be expressed with MDPs [Filar and Vrieze 1996; Puterman 1994]. In cases where we model the unknown behavior as stochastic (e.g., failures) and known properties of the environment as adversarial, $2^{1/2}$ player games are required. In such games, we have one player (the controller) playing against another one (the environment) and both interact with a third player who plays according to a certain probabilistic distribution.

3.4. Knowledge-Based Controllers

Conventional controllers (e.g., equation-based or adaptive controllers) can be designed following established mathematically grounded processes, guaranteeing the effectiveness of the controllers, including their stability, settling time, or robustness. However, a correct design and the understanding of the underlying theory may require specific

mathematical knowledge, not mastered by most software engineering. Furthermore, modeling and controlling systems exhibiting particularly complex nonlinear behaviors may require a good degree of expertise, since standard controller design processes seldom apply to them off the shelf.

Knowledge-based controllers allow overcoming the difficulties of formalizing both a system behavior and its control law. Indeed, a system behavior, including its knobs, its outputs, (optionally) its internal state, and the effects of each possible control action, are represented in a form suitable for formal deduction and inference by a reasoning engine. The reasoning engine is in charge, then, of deciding the most appropriate control actions based on the knowledge about the system augmented with monitoring information about the current state of execution. Due to the higher level of abstraction and often the lack of knowledge about the internal machinery of the reasoning engine, proving certain properties of these controllers might be harder [Levine 2010].

There are several possible choices for the reasoning engine and consequently for the representation of the knowledge base. In the following, we will briefly recall the basic principles behind two popular approaches: fuzzy rule-based reasoning and case-based reasoning.

3.4.1. Fuzzy Rule-Based Control. Fuzzy control is grounded in fuzzy logic, which is, in its simplest form, an extension of propositional logic where each variable may have a *degree of truth* ranging in the interval $[0, 1] \cap \mathbb{R}$ [Yager and Zadeh 2012]. The reasoning in this framework is based on fuzzy inference. Fuzzy inference is the process of mapping a set of control inputs to a set of control outputs through a set of fuzzy rules. The main application of fuzzy controllers is for types of problems that cannot be represented by explicit mathematical models due to high nonlinearity of the system. Instead, the potential of fuzzy logic lies in its capacity to approximate that nonlinearity by knowledge in a similar way to the human perception and reasoning. The explicit knowledge base (encompassing the user-defined fuzzy rules) component is one of the unique aspects of such type of controllers. Instead of sharp switching between modes based on thresholds, control output changes smoothly from different regions of behavior depending on the dominant rules.

Fuzzy controllers have been applied in the context of virtualized resource management [Xu et al. 2007; Rao et al. 2011; Lama and Zhou 2013] and cloud computing [Wang et al. 2015; Jamshidi et al. 2014]. For instance, the inputs to such controllers may include the workload level (w) and response time (rt) and the output may be the scaling action (sa) in terms of increment (or decrement) in the number of VMs. The design of a fuzzy controller, in general, involves the following tasks: (1) defining the fuzzy sets and membership functions of the input signals and (2) defining the rule base that determines the behavior of the controller in terms of control actions using the linguistic variables defined in the previous task. The very first step in the design process is to partition the state space of each input variable into various fuzzy sets through membership functions. Each fuzzy set is associated with a linguistic term such as "low" or " high." The membership function, denoted by $\mu_y(x)$, quantifies the degree of membership of an input signal x to the fuzzy set y (cf. Figure 3). As shown, three fuzzy sets have been defined for the workload to achieve a reasonable granularity in the input space while keeping the number of states small to reduce the set of rules in the knowledge base.

The next step consists of defining the inference machinery for the controller. Here we need to define elasticity policies in terms of rules: "`IF` (w is *high*) `AND` (rt is *bad*) `THEN` ($sa = +2$)," where the output function is a constant value. Note that depending on the problem at hand, this can be any finite discrete set of actions. For the definition of the functions in the rule consequents, the knowledge and experience of a human expert

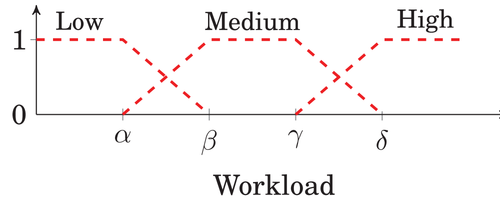

Fig. 3. Fuzzy membership functions for auto-scaling variables.

are generally used. In the situations where no a priori knowledge for defining such rules is assumed, a learning mechanism can instead be adopted [Lama and Zhou 2013; Jamshidi et al. 2016].

Once the fuzzy controller is designed, the execution of the controller is composed of three steps: (1) fuzzification of the inputs, (2) fuzzy reasoning, and (3) defuzzification of the output. Fuzzifier projects the crisp data onto fuzzy information using membership functions. Fuzzy engine reasons on information based on a set of fuzzy rules and derives fuzzy actions. Defuzzifier reverts the results back to crisp mode and activates an adaptation action. The output is calculated as a weighted average:

$$y(x) = \sum_{i=1}^{N} \mu_i(x) \times a_i, \tag{2}$$

where N is the number of rules, $\mu_i(x)$ is the firing degree of the rule i for the input signal x, and a_i is the consequent function for the same rule.

3.4.2. Case-Based Reasoning. Case-based reasoning (CBR) is based on the recognition of similar cases from previous experience [Qian et al. 2014]. The reasoning process comprises the following steps:

(1) Matching the new situation to prior cases
(2) Identifying the closest matches
(3) Combining the closest matches to find a solution
(4) Storing the new case and the solution in the case base

Given an initial set of cases, a CBR grows its experience while operating at runtime, making faster decisions for known cases by reusing what has "worked before" [Lambert 2001] instead of recomputing complex interpolations among the closest cases.

3.4.3. Learning Knowledge-Based Controllers. The lack of knowledge about the system or some of its parts during design may limit the effectiveness of a knowledge-based controller. For example, (de)fuzzification mechanisms can be retuned or new inference rules can be added to the controller after additional knowledge is gathered at runtime. Machine-learning techniques have been applied for this purpose. A notable example is the use of reinforcement learning to automatically build a rule set for a fuzzy controller based on the evaluation of explorative adaptation action selections [Tesauro 2007; Jamshidi et al. 2016; Lama and Zhou 2013].

3.5. Dealing with Uncertainty in Control Strategies for Adaptation

Independently of the particular model-based control technique that is adopted, the first task in controller design is modeling the system to be controlled. This system, however, may be very complex and its dynamics may not be completely understood. Developing accurate system models over an operating range is a challenging task [Åström and Murray 2008], especially for computing systems [Hellerstein et al. 2004; Maggio et al.

2012; Leva et al. 2013; Papadopoulos et al. 2015]. Even if a detailed mathematical model is available, it may be complex and make the controller design challenging and computationally expensive [Morari and Zafiriou 1989]. Of course, the more complete and accurate the model, the more robust the controller is. Thus, there is tension between the complexity of the model and the controller's robustness to unmodeled or uncertain characteristics.

Besides the challenge of determining an appropriate level of abstraction to balance the simplicity of the model and the amount of information available to the controller, engineering self-adaptive software requires a peculiar attention to modeling of uncertain phenomena. Uncertainty can be at the same time the driver for and the outcome of adding self-adaptation capabilities to a system. The research community devolved extensive efforts to studying and modeling uncertainty in self-adaptive software (see, e.g., Esfahani and Malek [2013], Esfahani et al. [2011], Cheng and Garlan [2007], Ramirez et al. [2012a, 2012b], Giese et al. [2014], Bencomo and Belaggoun [2014], Alessia Knauss et al. [2016], and Cheng et al. [2009]) and elicited three main sources for it:

(1) Requirements can change over time and have different priorities for different users [Baresi et al. 2006]; specifications are incomplete, and designs can intentionally be left open ended [Letier et al. 2014].
(2) Software interacting with physical systems, including unreliable hardware infrastructures, has possibly to deal with an epistemic uncertainty intrinsic in some physical phenomena (e.g., a GPS localization device outputs a distribution over likely positions) [Zhang et al. 2016].
(3) Software deployed in the wild has to tackle interactions with a variety of users, whose behavior is hardly predictable and often changes over time [Cámara et al. 2015].

On the other hand, the very addition of adaptation capabilities to software, if not properly designed, may introduce additional uncertainty about its behavior. This may compromise the overall dependability of the self-adaptive systems [Calinescu et al. 2012] and is the primary driver for developing principled design techniques and quality assurance practices. We so far described the mathematical principles that control theory offers to design *reliable-by-design* controllers.

In the remainder of this section, we will first discuss three common ways for dealing with uncertainty in control theoretical software adaptation, and then elicit a set of tasks software engineers can be required to perform when dealing with uncertainty.

Dealing with uncertainty in control. There are different ways to deal with uncertainty, including *adaptive and robust control*, *probability theory*, and *fuzzy logic*.

In *classical control* theory, the typical approach is to provide a model, possibly probabilistic, of the uncertainty and to adopt a control technique that is conceived to be robust against the modeled uncertainty. Possible approaches are then to deal with the uncertainty following a worst-case scenario approach [Campi et al. 2009] (e.g., this is the case of $\mathcal{H}_\infty$ control) [Skogestad and Postlethwaite 2007] or to adapt the controller's behavior online, according to measured quantities on the system (e.g., adaptive control) [Åström and Wittenmark 2013].

When the uncertainty can be restricted to a known interval, one can also use interval Markov models. Recently, several techniques have been developed [Chatterjee et al. 2008; Benedikt et al. 2013; Chen et al. 2013; Puggelli et al. 2013]. Typically, these methods work by determining the extremal values a probabilistic parameter can take. When the range of uncertainty is not known, one can opt for parametric Markov models that construct a symbolic model that can later be evaluated with specific probabilities.

Recent work includes methods from parametric model checking [Hahn et al. 2010; Filieri et al. 2011a; Dehnert et al. 2015].

Finally, *fuzzy logic* controllers deal with uncertainty quantifying the ambiguity due to the unmodeled underlying phenomena [Jamshidi et al. 2014]. Controllers based on fuzzy theory are called fuzzy logic controllers [Jantzen 2013]. They are particularly suited for nonlinear control and nonprobabilistic uncertainty [Mendel and Wu 2010].

Common tasks in dealing with uncertainty. The fact that many different approaches exist to deal with uncertainty is a figure of how difficult this problem is. In fact, dealing with uncertainty raises a number of different challenges:

—*Model selection.* To enable tractable analysis, most mathematical models are inherently approximate. While many phenomena can be modeled using probability theory or analyzed using fuzzy logic, software designers must select appropriate models.
—*Knowledge evolution.* Typically adaptive software systems use some sort of adaptation knowledge either in terms of analytical model or explicit adaptation rule for controlling the underlying software systems. However, the model may drift at runtime due to the changes in the assumptions on which the model is made. Some recent work [Abbas et al. 2011; Jamshidi et al. 2016] investigated this challenge.
—*Reasoning about lack of domain knowledge.* Explicitly adding uncertainty to software is an application-specific effort. Generic techniques for trading application speedup and quality are still in their early stages. There are some solutions that address uncertainty by learning parameters online. For instance, reinforcement learning can explore the environment and pass knowledge to a controller for more effective decision making [Hoffmann 2015].
—*Reasoning about multiple sources of uncertainty.* Even when individual uncertainty sources are modeled using a single underlying theory (e.g., probability theory), it is not clear how to combine different models to characterize the uncertainty of the computation. Currently, combining multiple uncertainty models in a single application requires careful, application-specific approaches. General methods for combining uncertainty models is another key challenge.
—*System adaptation.* Given accuracy characterization of the approximate programs, runtime systems can use this information for accuracy, performance, and energy targets [Baek and Chilimbi 2010; Hoffmann et al. 2011; Samadi et al. 2014]. However, adaptively controlling accuracy in a systematic manner is still often an ad hoc process. The work on more general methodologies is ongoing [Filieri et al. 2014; Hoffmann 2014].

4. SOFTWARE ENGINEERING FOR CONTROL THEORY

The mathematical foundation of control theory has shaped its practical development: the standard practice in the design and implementation of control systems is to devise an ad hoc solution mathematically for each problem instance [Levine 2010] disregarding, in a first place, implementation concerns. Whereas the mathematical derivation of tailored controllers for complex problems may still require human expertise, software engineering established principles and design techniques that are carving their way in the design of control systems. In this section, we will sketch some of the directions in which software engineering is shaping the design and implementation of software controllers. These results can also ease the integration of control in self-adaptive software.

Domain-specific languages. The established approach to controller design and implementation is the use of domain-specific languages. Development suites like Matlab and Simulink, Modelica, and Ptolemy II [Eker et al. 2003], as well as less popular ones like ACTRESS [Krikava et al. 2014] or EUREMA [Vogel and Giese 2014], provide

domain-specific constructs to ease the formalization of dynamic models and automatically generate executable code targeted at different hardware infrastructures. Despite the success of these tools, the evolution of both sophisticated control techniques and execution platforms is presenting unprecedented challenges. As an example, model predictive control requires the solution of complex optimization problems during runtime. Programmable and specialized hardware presents new opportunities for automatically transforming and compiling domain-specific languages into platform-specific ones, unleashing computational power beyond the reach of general-purpose architectures. While preliminary ad hoc solutions exist (e.g., Kerrigan [2014] and Hartley et al. [2014]), their generalization is still a concern for researchers in software engineering and programming languages [Dubach et al. 2012]. Just-in-time compilation, generative programming, and modular staging [Rompf and Odersky 2010] are growing in importance in software engineering and can change the shape of controller development, as early results demonstrate [Ofenbeck et al. 2013; Kong et al. 2013].

Design patterns. Control theory developed common solution patterns for several complex problems, usually involving multiple controllers. This goes under the term of " control allocation strategies" [Levine 2010], and such strategies have clear semantic similarities to design patterns in software engineering. The growing scale and complexity of industrial control [Vyatkin 2013] are calling for control engineers to face new software engineering problems and to start exploring specific design patterns [Sanz and Zalewski 2003; Scattolini 2009]. To further improve in this direction, a deeper cross-fertilization between software and control engineering is needed. On the other hand, robotics and cyber-physical systems are offering a natural playground for this encounter [Brugali 2007; Rajkumar et al. 2010].

Verification and validation of controller implementation. Although control theory developed techniques to validate system models experimentally and in turn the theoretical effectiveness of their controllers, implementation may differ from its design. Software engineering is playing a role in both producing correct-by-construction implementations from the mathematical description of controllers [Kowshik et al. 2002; Darulova and Kuncak 2014; Cai 2002] and defining specialized testing and verification procedures [Darulova and Kuncak 2013; Ismail et al. 2015; Matinnejad et al. 2016; Zuliani et al. 2010; Villegas et al. 2011; Camara et al. 2013].

Implementation of distributed control. Networked control systems are rapidly developing thanks to advances in both industrial automation and robotics [Yang 2006]. Software engineering developments in the fields of actor models and reactive programming are supporting their implementation Liu et al. [2004], Muscholl [2015], and Delaval et al. [2013]. However, to the best of our knowledge, no languages specific for distributed control have been developed so far.

5. CONCLUSIONS AND OPEN RESEARCH CHALLENGES

Self-adaptation mechanisms for software systems require a formal grounding to ensure their dependability and effectiveness. Control theory showed promising results for providing such grounding in a variety of situations. In this article, we elicited and discussed a set of control strategies software engineering can glean from to define dependable software adaptation techniques. Furthermore, we presented control theoretical results under the light of common software development activities, from requirements formalization to implementation and verification, to show how an understanding of each of those activities can benefit from the understanding of control concepts.

Although not all software adaptation problems find a straightforward casting into the control domain, the concepts and techniques proposed in this article can represent

a core toolkit enriching the corpus of established solutions for self-adaptive software design.

To conclude this work, in the reminder of this section, we elicit a first set of open research challenges in developing *controllable software*. We divide these into two categories: challenges for software engineers and challenges for control engineers. For each category, we first describe the overarching goal and then some key objectives that must be met to achieve this goal.

5.1. The Software Engineering Perspective

Software engineers must create *controllable software*, that is, make control a first-class design concern [Müller et al. 2008; Brun et al. 2009]. Software engineers will benefit as controllable software permits formal analysis of its dynamic behavior. For example, engineers who design controllable software can then mathematically reason about its performance, energy, security, reliability, or other nonfunctional requirements—despite the unpredictability and changeability of the environment.

The need for *adaptive* software that can change its own behavior in a changing environment has long been recognized [Cheng et al. 2009]. Controllable software is a subset of adaptive software. While adaptive software can change its behavior, controllable software changes its behavior according to a set of well-founded mathematical models whose dynamics can be formally analyzed. This section presents several objectives that will help achieve the goal of creating controllable software.

Control design patterns. To make control techniques directly usable to software engineers—without requiring a deep knowledge of control theory—the community can work on identifying design patterns specific to control. This would allow for the definition of reusable solutions and easy communication of design choices; capturing the similarities within classes of software adaptation problems solvable with control techniques would allow a faster adoption of control from the early design stages. A top-down approach to the definition of control design patterns would first identify controllers and then determine the applications to which they are best suited (e.g., Filieri et al. [2014, 2015a]). A bottom-up approach would first find applications with feedback, then apply control techniques, and finally generalize (e.g., Klein et al. [2014]).

Control smells. A *code smell* is "any symptom in the source code of a program that possibly indicates a deeper problem" [Fowler and Beck 1999]. While code smells are not bugs by themselves, they may indicate a design weakness. When a code smell is discovered, an engineer must decide whether or not to redesign. Categorizing common code smells and remedies allows engineers to focus on those parts of the program that are more likely to cause problems. Analogously, control smells may indicate where in the code it would be possible to add control solutions to improve a software ability of reaching its goals. Automatic knob identification is an example of such applications [Harman et al. 2014]. Furthermore, control smells can point to erroneous or misplaced implementation of controllers. For example, implementing two uncoordinated controllers targeting the same goal but using different actuators may lead to oscillating behavior; if the problem is detected, a remedy might be revising the control allocation or devising a multiple-output controller.

Testing and debugging controlled systems. A self-adaptive software system's peculiar reactions to different environmental conditions challenge traditional quality assurance approaches. In Section 2.7, we reviewed recent results on validation and verification of controlled systems. However, controllers alter the behavior of the system, possibly leading to emerging behaviors not considered when analyzing the system alone. Verification and validation, as well as testing, of the controlled system has to

take into account the interactions between the controller, the system, and its environment. How to reproduce a self-adaptive system's execution, how to identify an error's source, and how to debug it are just examples of the challenges controllable software presents to software engineering.

Making nonadaptive software adaptive. While design for controllability is arguably a long-term goal, empowering existing software with adaptation capabilities is a short-term goal, which would improve its robustness and scalability. Most current software is not adaptive. Enabling adaptation requires identifying both actuators that alter behavior and sensors that measure this behavior, as well as devising controllers for each actuator. The identified sensors and actuators have to guarantee the observability and controllability of the system. Observability is the property that the system state is inferable from the sensor measurements; controllability is the property that the available actuators can drive the system toward its goals. While in control theory these concepts are formalized and have a mathematical counterpart for many classes of dynamic models, it is still unclear how to assess these properties for software artifacts.

Automatic synthesis and update of controllers. Most software is not designed with controllability in mind. While in a long-term vision controllability may become a first-class concept in software design, (semi-)automated controller synthesis can provide a suitable solution for a variety of problems. Though no general synthesis method is currently known, several classes of software control problems are addressed with automated synthesis techniques (e.g., Filieri et al. [2014], D'ippolito et al. [2013], Filieri et al. [2011b], and Hoffmann [2014]. Automatic learning for knowledge-based controllers [Tesauro 2007] and auto-tuning mechanisms tailoring general solutions to specific cases are other means allowing practitioners to apply control theoretical adaptation mechanisms without mastering the underlying mathematical knowledge. This thread of research can empower practitioners with broadly applicable, ready-to-use solutions for several adaptation problems.

Coordinating multiple controllers. Complex software systems may require multiple controllers, possibly of different types. For example, functional adaptation can be managed by event-based controllers, able to select different component assemblies to guarantee safety or liveness. Quantitative goals can be handled by equation-based controllers. Multiple controllers may interact, introducing undesirable emerging behaviors, possibly compromising system stability. Analogous problems occur in assembly of off-the-shelf components that may embed self-adaptation capabilities. Even self-adaptive execution infrastructures may interfere with the behavior of self-adaptive software. Examples are cloud-based execution environments, but also modern CPUs, such as the turbo-boost capabilities of some Intel processors able to automatically adapt the degree of parallelism and frequency of the cores. The coordination of multiple controllers is an open challenge for both software engineering and control theory. Several results from the analysis of distributed and concurrent systems have been adapted for event-based controllers; however, these results can hardly cope with heterogeneous controllers.

5.2. The Control Engineering Perspective

Control theory has developed a broad set of mathematically grounded techniques for the control of physical processes. While several of these techniques have been adapted to different domains, software systems present unprecedented challenges to control solutions. In particular, many software behaviors are arguably not bounded to the law of the physical world, requiring control engineers to develop new modeling, sensing, actuation, and control techniques. While this development is most likely a joint

effort of software and control engineers to design controllable software on a first-hand basis, control engineers are challenged with the need of changing their perspective on several established problems.

Software models for control. Software systems typically lack the mathematical models as described in Section 2.3. Many of the control methods described in Section 3 require a model in terms of a differential or difference equations that software engineers would not normally define. To model software systems mathematically, the numerous methods from *system identification* can be used [Ljung 2012]. Broadly, these methods construct models using measurements. For software, building these models might be easier than for physical systems as running software is usually cheaper and can be done faster. This means large amounts of data for identification purposes are easily available. Nonetheless, system identification may be ineffective without the preliminary definition of a class of models to be trained from data. Software behaviors may be difficult to cast into established classes of differential or difference equations, possibly requiring the definition of more complex classes of hybrid models.

Monitoring and instrumentation. While it can be prohibitively expensive to add a sensor to an existing physical system, it is relatively easy to add some code to a piece of software to enable another measurement of its internal state. This can be a "shortcut" toward achieving certain control goals that for physical systems would require extensive filter design or complex control strategies. On the other hand, it requires the development of techniques for the possibly optimal instrumentation of the code not only to satisfy control needs (e.g., observability) but also to take into account the impact of the monitoring infrastructure on the production systems, in terms of both effectiveness and overhead.

Software actuation. In general, physical plants that need control strategies are designed to be controlled, according to the natural constraints imposed by physics. Pumps and valves are placed following standard or established industrial design processes. Such standard processes do not exist for controllable software and need to be defined. Similarly to the design of monitoring infrastructures, the optimal design of actuators has to take into account both control needs (e.g., controllability) and software production needs, again in terms of effectiveness but also in terms of impact on the global software design and cost of actuation.

REFERENCES

N. Abbas, J. Andersson, and D. Weyns. 2011. Knowledge evolution in autonomic software product lines. In *Proceedings of the 15th International Software Product Line Conference (SPLC'11)*. ACM Press, New York, NY, 1.

T. Abdelzaher, Y. Diao, J. Hellerstein, C. Lu, and X. Zhu. 2008. Introduction to control theory and its application to computing systems. In *Performance Modeling and Engineering*, Zhen Liu and Cathy H. Xia (Eds.). Springer US, 185–215.

A. Knauss, D. Damian, X. Franch, A. Rook, H. A. Mller, and A. Thomo. 2016. ACon: A learning-based approach to deal with uncertainty in contextual requirements at runtime. *Information and Software Technology* 70 (2016), 85–99.

R. Alur. 2011. Formal verification of hybrid systems. In *2011 Proceedings of the International Conference on Embedded Software (EMSOFT'11)*. 273–278.

Z. Artstein. 1980. Discrete and continuous bang-bang and facial spaces or: Look for the extreme points. *SIAM Review* 22, 2 (1980), 172–185.

K. Åström. 2008. Event based control. In *Analysis and Design of Nonlinear Control Systems*, A. Astolfi and L. Marconi (Eds.). Springer, Berlin, 127–147.

K. Åström and T. Hägglund. 2006. *Advanced PID Control*. ISA-The Instrumentation, Systems, and Automation Society, Research Triangle Park, NC.

K. Åström and R. Murray. 2008. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press.

K. Åström and B. Wittenmark. 2013. *Adaptive Control*. Dover Publications.

W. Baek and T. M. Chilimbi. 2010. Green: A framework for supporting energy-conscious programming using controlled approximation. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*. ACM, New York, NY, 198–209.

C. Baier, M. Grosser, M. Leucker, B. Bollig, and F. Ciesinski. 2004. Controller synthesis for probabilistic systems. In *Proceedings of the 3rd International Conference on Theoretical Computer Science (IFIP/TCS)*. Springer, 493–506.

S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. 2004. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering* 30, 5 (May 2004), 295–310.

L. Baresi, E. D. Nitto, and C. Ghezzi. 2006. Toward open-world software: Issues and challenges. *Computer* 39, 10 (Oct. 2006), 36–43.

T. Batista, A. Joolia, and G. Coulson. 2005. Managing dynamic reconfiguration in component-based systems. In *Proceedings of the 2nd European Conference on Software Architecture (EWSA'05)*. Springer-Verlag, Berlin, 1–17.

A. Bemporad and M. Morari. 1999. Robust model predictive control: A survey. In *Robustness in Identification and Control*, A. Garulli and A. Tesi (Eds.). Lecture Notes in Control and Information Sciences, Vol. 245. Springer, London, 207–226.

N. Bencomo and A. Belaggoun. 2014. A world full of surprises: Bayesian theory of surprise to quantify degrees of uncertainty. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 460–463.

M. Benedikt, R. Lenhardt, and J. Worrell. 2013. LTL model checking of interval markov chains. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 32–46.

S. Boyd, C. Baratt, and S. Norman. 1990. Linear controller design: Limits of performance via convex optimization. *Proceedings of the IEEE* 78, 3 (March 1990), 529–574.

V. A. Braberman, N. D'Ippolito, J. Kramer, D. Sykes, and S. Uchitel. 2015. MORPH: A reference architecture for configuration and behaviour self-adaptation. In *1st International Workshop on Control Theory for Software Engineering (CTSE'15)*.

T. Brázdil, V. Forejt, and A. Kučera. 2008. Controller synthesis and verification for markov decision processes with qualitative branching time objectives. *Lecture Notes in Computer Science*, Vol. 5126. Springer, 148–159.

F. Brosig, P. Meier, S. Becker, A. Koziolek, H. Koziolek, and S. Kounev. 2015. Quantitative evaluation of model-driven performance analysis and simulation of component-based architectures. *IEEE Transactions on Software Engineering* 41, 2 (Feb. 2015), 157–175.

D. Brugali. 2007. *Software Engineering for Experimental Robotics*. Springer, Berlin.

Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. 2009. Engineering self-adaptive systems through feedback loops. In *Software Engineering for Self-Adaptive Systems*. Lecture Notes in Computer Science, Vol. 5525. Springer, 48–70.

K.-Y. Cai. 2002. Optimal software testing and adaptive software testing in the context of software cybernetics. *Information and Software Technology* 44, 14 (2002), 841–855.

R. Calinescu, C. Ghezzi, M. Kwiatkowska, and R. Mirandola. 2012. Self-adaptive software needs quantitative verification at runtime. *Communications of the ACM* 55, 9 (2012), 69–77.

A. Camacho, P. Marti, M. Velasco, C. Lozoya, R. Villa, J. M. Fuertes, and E. Griful. 2010. Self-triggered networked control systems: An experimental case study. In *2010 IEEE International Conference on Industrial Technology (ICIT'10)*. 123–128.

E. Camacho and C. Alba. 2013. *Model Predictive Control*. Springer, London.

J. Camara, R. de Lemos, N. Laranjeiro, R. Ventura, and M. Vieira. 2013. Robustness evaluation of controllers in self-adaptive software systems. In *2013 6th Latin-American Symposium on Dependable Computing (LADC'13)*, 1–10.

J. Cámara, G. A. Moreno, and D. Garlan. 2015. Reasoning about human participation in self-adaptive systems. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'15)*. IEEE Press, Piscataway, NJ, 146–156.

M. C. Campi and S. Garatti. 2011. A sampling-and-discarding approach to chance-constrained optimization: Feasibility and optimality. *Journal of Optimization Theory and Applications* 148, 2 (2011), 257–280.

M. C. Campi, S. Garatti, and M. Prandini. 2009. The scenario approach for systems and control design. *Annual Reviews in Control* 33, 2 (2009), 149–157.

K. Chatterjee, T. A. Henzinger, B. Jobstmann, and R. Singh. 2015. Measuring and synthesizing systems in probabilistic environments. *Journal of the ACM* 62, 1, Article 9 (March 2015), 34 pages.

K. Chatterjee, M. Jurdziński, and T. A. Henzinger. 2003. Simple stochastic parity games. In *Computer Science Logic*, Matthias Baaz and Johann A. Makowsky (Eds.). Lecture Notes in Computer Science, Vol. 2803. Springer, Berlin, 100–113.

K. Chatterjee, M. Jurdziński, and T. A. Henzinger. 2004. Quantitative stochastic parity games. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'04)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 121–130.

K. Chatterjee, K. Sen, and T. A. Henzinger. 2008. Model-checking ω-regular properties of interval markov chains. In *Foundations of Software Science and Computational Structures*. Springer, 302–317.

T. Chen, T. Han, and M. Kwiatkowska. 2013. On the complexity of model checking interval-valued discrete time Markov chains. *Information Processing Letters* 113, 7 (2013), 210–216.

B. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. Kienle, J. Kramer, M. Litoiu, S. , R. Mirandola, H. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle. 2009. Software engineering for self-adaptive systems. In *Software Engineering for Self-Adaptive Systems*. Springer-Verlag, Berlin, 1–26.

S.-W. Cheng and D. Garlan. 2007. Handling uncertainty in autonomic systems. In *Proceedings of the International Workshop on Living with Uncertainties (IWLU'07), co-located with the 22nd International Conference on Automated Software Engineering (ASE'07)*.

E. Darulova and V. Kuncak. 2013. *Certifying Solutions for Numerical Constraints*. Springer, Berlin, 277–291.

E. Darulova and V. Kuncak. 2014. Sound compilation of reals. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14)*. ACM, New York, NY, 235–248.

E. M. Dashofy, A. van der Hoek, and R. N. Taylor. 2002. Towards architecture-based self-healing systems. In *Proceedings of the 1st Workshop on Self-Healing Systems (WOSS'02)*. ACM, New York, NY, 21–26.

C. Dehnert, S. Junges, N. Jansen, F. Corzilius, M. Volk, H. Bruintjes, J.-P. Katoen, and E. Ábrahám. 2015. *PROPhESY: A PRObabilistic ParamEter SYnthesis Tool*. Springer, 214–231.

G. Delaval, E. Rutten, and H. Marchand. 2013. Integrating discrete controller synthesis into a reactive programming language compiler. *Discrete Event Dynamic Systems* 23, 4 (2013), 385–418.

Y. Diao, J. Hellerstein, S. Parekh, R. Griffith, G. Kaiser, and D. Phung. 2005. Self-managing systems: A control theory foundation. In *ECBS Workshop*. 441–448.

Y. Diao, J. L. Hellerstein, S. Parekh, R. Griffith, G. E. Kaiser, and D. Phung. 2006. A control theory foundation for self-managing computing systems. *IEEE Journal of Selected Areas of Communications* 23, 12 (Sept. 2006), 2213–2222.

Z. Ding, Y. Zhou, and M. Zhou. 2014. Modeling self-adaptive software systems with learning petri nets. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion'14)*. ACM, New York, NY, 464–467.

N. D'ippolito, V. Braberman, N. Piterman, and S. Uchitel. 2013. Synthesising non-anomalous event-based controllers for liveness goals. *ACM Tranactions on Software Engineering Methodologies* 22, Article 9 (2013), 36 pages.

N. R. D'Ippolito, V. Braberman, N. Piterman, and S. Uchitel. 2010. Synthesis of live behaviour models. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10)*. ACM, New York, NY, 77–86.

R. Dorf and R. Bishop. 2008. *Modern Control Systems*. Prentice Hall.

C. Dubach, P. Cheng, R. Rabbah, D. F. Bacon, and S. J. Fink. 2012. Compiling a high-level language for GPUs: (Via language support for architectures and compilers). In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*. ACM, New York, NY, 1–12.

J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. 2003. Taming heterogeneity-the ptolemy approach. *Proceedings of the IEEE* 91, 1 (2003), 127–144.

I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli. 2009. Model evolution by run-time parameter adaptation. In *International Conference on Software Engineering*. IEEE, 111–121.

N. Esfahani, E. Kouroshfar, and S. Malek. 2011. Taming uncertainty in self-adaptive software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (SIGSOFT/FSE'11)*. ACM, 234–244.

N. Esfahani and S. Malek. 2013. Uncertainty in self-adaptive software systems. In *Software Engineering for Self-Adaptive Systems II*, Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary Shaw (Eds.). Vol. 7475. Springer, Berlin, 214–238.

J. Filar and K. Vrieze. 1996. *Competitive Markov Decision Processes*. Springer-Verlag New York, New York, NY.

A. Filieri, C. Ghezzi, A. Leva, and M. Maggio. 2011b. Self-adaptive software meets control theory: A preliminary approach supporting reliability requirements. In *26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11),* Perry Alexander, Corina S. Pasareanu, and John G. Hosking (Eds.). IEEE, 283–292.

A. Filieri, C. Ghezzi, and G. Tamburrelli. 2011a. Run-time efficient probabilistic model checking. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. ACM, New York, NY, 341–350.

A. Filieri, H. Hoffmann, and M. Maggio. 2014. Automated design of self-adaptive software with control-theoretical formal guarantees. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. ACM, New York, NY, 299–310.

A. Filieri, H. Hoffmann, and M. Maggio. 2015a. Automated multi-objective control for self-adaptive software design. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, 13–24.

A. Filieri, M. Maggio, K. Angelopoulos, N. D'Ippolito, I. Gerostathopoulos, A. Hempel, H. Hoffmann, P. Jamshidi, E. Kalyvianaki, C. Klein, F. Krikava, S. Misailovic, A. Papadopoulos, S. Ray, A. Shariflloo, S. Shevtsov, M. Ujma, and T. Vogel. 2015b. Software engineering meets control theory. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'15)*. IEEE, 12.

M. Fowler and K. Beck. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

M. Franzle and C. Herde. 2007. HySAT: An efficient proof engine for bounded model checking of hybrid systems. *Formal Methods in System Design* 30, 3 (2007), 179–198.

S. Gao, S. Kong, and E. Clarke. 2013. Satisfiability modulo ODEs. In *Formal Methods in Computer-Aided Design (FMCAD'13)*. 105–112.

D. Garlan, S. Cheng, A. Huang, B. R. Schmerl, and P. Steenkiste. 2004a. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer* 37, 10 (2004), 46–54.

D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. 2004b. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer* 37, 10 (Oct. 2004), 46–54.

H. Giese, N. Bencomo, L. Pasquale, A. J. Ramirez, P. Inverardi, S. Waetzoldt, and S. Clarke. 2014. Living with uncertainty in the age of runtime models. In *Models@run.time: Foundations, Applications, and Roadmaps*. Springer, 47–100.

R. Goebel, R. Sanfelice, and A. Teel. 2012. *Hybrid Dynamical Systems: Modeling, Stability, and Robustness*. Princeton University Press.

G. C. Goodwin, H. Kong, G. Mirzaeva, and M. M. Seron. 2014. Robust model predictive control: Reflections and opportunities. *Journal of Control and Decision* 1, 2 (April 2014), 115–148.

S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam. 2011. Cuanta: Quantifying effects of shared on-chip resource interference for consolidated virtual machines. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC'11)*. ACM, New York, NY, Article 22, 14 pages.

L. Grüne and J. Pannek. 2011. *Nonlinear Model Predictive Control*. Springer.

E. M. Hahn, H. Hermanns, B. Wachter, and L. Zhang. 2010. PARAM: A model checker for parametric markov models. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV'10) (LNCS)*, Vol. 6174. Springer, 660–664.

M. Harman, Y. Jia, W. B. Langdon, J. Petke, I. H. Moghadam, S. Yoo, and F. Wu. 2014. Genetic improvement for adaptive software engineering (keynote). In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)*.

E. N. Hartley, J. L. Jerez, A. Suardi, J. M. Maciejowski, E. C. Kerrigan, and G. A. Constantinides. 2014. Predictive control using an FPGA with application to aircraft control. *IEEE Transactions on Control Systems Technology* 22, 3 (May 2014), 1006–1017.

W. P. M. H. Heemels, K. H. Johansson, and P. Tabuada. 2012. An introduction to event-triggered and self-triggered control. In *2012 IEEE 51st Annual Conference on Decision and Control (CDC'12)*. 3270–3285.

W. P. M. H. Heemels, J. H. Sandee, and P. P. J. Van Den Bosch. 2008. Analysis of event-driven controllers for linear systems. *International Journal of Control* 81, 4 (2008), 571–590.

J. L. Hellerstein. 2009. Engineering autonomic systems. In *Proceedings of the 6th International Conference on Autonomic Computing (ICAC'09)*. ACM, New York, NY, 75–76.

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

J. L. Hellerstein, V. Morrison, and E. Eilebrecht. 2010. Applying control theory in the real world: Experience with building a controller for the .NET thread pool. *SIGMETRICS Performance Evaluation Review* 37, 3 (Jan. 2010), 38–42.

T. Henzinger, P.-H. Ho, and H. Wong-Toi. 1997. HyTech: A model checker for hybrid systems. In *Computer Aided Verification*. Lecture Notes in Computer Science, Vol. 1254. Springer, Berlin, 460–463.

T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. 1998. What's decidable about hybrid automata? *Journal of Computer and System Sciences* 57, 1 (1998), 94–124.

H. Hermes and J. P. Lasalle. 1969. *Functional Analysis and Time Optimal Control*. Elsevier Science.

H. Hoffmann. 2014. CoAdapt: Predictable behavior for accuracy-aware applications running on power-aware systems. In *2014 26th Euromicro Conference on Real-Time Systems (ECRTS'14)*. IEEE, 223–232.

H. Hoffmann. 2015. Software engineering meets control theory. In *Proceedings of the 25th International Symposium on Operating Systems (SOSP'15)*. ACM, 14.

H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. 2011. Dynamic knobs for responsive power-aware computing. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*. 14.

P. Inverardi and M. Tivoli. 2003. Software architecture for correct components assembly. In *Formal Methods for Software Architectures*. Springer, 92–121.

H. I. Ismail, I. V. Bessa, L. C. Cordeiro, E. B. de Lima Filho, and J. E. Chaves Filho. 2015. *DSVerifier: A Bounded Model Checking Tool for Digital Systems*. Springer International Publishing, Cham, 126–131.

P. Jamshidi, A. Ahmad, and C. Pahl. 2014. Autonomic resource provisioning for cloud-based software. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'14)*. ACM, New York, NY, 95–104.

P. Jamshidi, A. Sharifloo, C. Pahl, H. Arabnejad, A. Metzger, and G. Estrada. 2016. Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures. In *Proceedings of the 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*. 70–79.

J. Jantzen. 2013. *Foundations of Fuzzy Control: A Practical Approach*. John Wiley & Sons.

S. Kang and D. Garlan. 2014. Architecture-based planning of software evolution. *International Journal of Software Engineering and Knowledge Engineering* 24, 2 (2014), 211–242.

J. Kephart and D. Chess. 2003. The vision of autonomic computing. *Computer* 36, 1 (Jan. 2003), 41–50.

E. C. Kerrigan. 2014. Co-design of hardware and algorithms for real-time optimization. In *2014 European Control Conference (ECC'14),*. 2484–2489.

H. Khalil. 2015. *Nonlinear Control*. Pearson Education.

C. Klein, M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez. 2014. Brownout: Building more robust cloud applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. ACM, New York, NY, 700–711.

M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan. 2013. When polyhedral transformations meet SIMD code generation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*. ACM, New York, NY, 127–138.

S. Kowshik, D. Dhurjati, and V. Adve. 2002. Ensuring code safety without runtime checks for real-time control systems. In *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'02)*. ACM, New York, NY, 288–297.

J. Kramer and J. Magee. 2007. Self-managed systems: An architectural challenge. In *International Conference on Software Engineering (ISCE'07), Workshop on the Future of Software Engineering (FOSE'07)*. 259–268.

F. Krikava, P. Collet, R. France, and others. 2014. ACTRESS: Domain-specific modeling of self-adaptive software architectures. In *Symposium on Applied Computing, Track on Dependable and Dependable and Adaptive Distributed Systems*.

P. R. Kumar and P. Varaiya. 1986. *Stochastic Systems: Estimation, Identification and Adaptive Control*. Prentice-Hall, Upper Saddle River, NJ.

M. Kwiatkowska and D. Parker. 2013. Automated verification and strategy synthesis for probabilistic systems. In *Proceedings of the 11th International Symposium on Automated Technology for Verification and Analysis (ATVA'13) (LNCS)*, Vol. 8172. Springer, 5–22.

G. Labinaz, M. M. Bayoumi, and K. Rudie. 1997. A survey of modeling and control of hybrid systems. *Annual Reviews in Control* 21 (1997), 79–92.

P. Lama and X. Zhou. 2013. Autonomic provisioning with self-adaptive neural fuzzy control for percentile-based delay guarantee. *Transactions on Autonomous and Adaptive Systems (TAAS)* 8, 2 (2013), 9.

S. Lambert. 2001. Knowledge-based control systems. In *Theory and Practice of Informatics (SOFSEM'01)*, Leszek Pacholski and Peter Ružička (Eds.). LNCS, Vol. 2234.

E. Letier, D. Stefan, and E. T. Barr. 2014. Uncertainty, risk, and information value in software requirements and architecture. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. ACM, New York, NY, 883–894.

A. Leva, M. Maggio, A. Papadopoulos, and F. Terraneo. 2013. *Control-Based Operating System Design*. IET.

A. Leva and A. Papadopoulos. 2013. Tuning of event-based industrial controllers with simple stability guarantees. *Journal of Process Control* 23, 9 (2013), 1251–1260.

W. Levine. 2010. *The Control Systems Handbook: Control System Advanced Methods,* (2nd ed). Taylor and Francis.

D. Liberzon. 2003. *Switching in Systems and Control*. Birkhäuser Boston.

J. Liu, J. Eker, and J. Janneck. 2004. Actor-oriented control system design: A responsible framework perspective. *IEEE Transactions on Control Systems Technology* 12, 2 (2004), 250–262.

L. Ljung. 2012. *System Identification: Theory for the User* (2nd ed.). Prentice Hall PTR, Upper Saddle River, NJ.

J. Lunze and F. Lamnabhi-Lagarrigue. 2009. *Handbook of Hybrid Systems Control: Theory, Tools, Applications*. Cambridge University Press.

J. Lunze and D. Lehmann. 2010. A state-feedback approach to event-based control. *Automatica* 46, 1 (2010), 211–215.

J. Maciejowski. 2002. *Predictive Control: With Constraints*. Pearson Education.

M. Maggio, H. Hoffmann, A. Papadopoulos, J. Panerati, M. Santambrogio, A. Agarwal, and A. Leva. 2012. Comparison of decision making strategies for self-optimization in autonomic computing systems. *ACM Transactions on Autonomous and Adaptive Systems* 7, 4, Article 36 (Dec. 2012), 32 pages.

R. Matinnejad, S. Nejati, L. C. Briand, and T. Bruckmann. 2016. Automated test suite generation for time-continuous simulink models. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. ACM, New York, NY, 595–606.

J. Mendel and D. Wu. 2010. *Perceptual Computing: Aiding People in Making Subjective Judgments*. Wiley-IEEE Press.

M. Morari and E. Zafiriou. 1989. *Robust Process Control*. Prentice Hall, Englewood Cliffs, NJ.

H. Müller, M. Pezzè, and M. Shaw. 2008. Visibility of control in adaptive systems. In *Proceedings of the 2nd International Workshop on Ultra-Large-Scale Software-intensive Systems (ULSSIS'08)*. ACM, New York, NY, 23–26.

A. Muscholl. 2015. *Automated Synthesis of Distributed Controllers*. Springer, Berlin, 11–27.

G. Ofenbeck, T. Rompf, A. Stojanov, M. Odersky, and M. Püschel. 2013. Spiral in scala: Towards the systematic construction of generators for performance libraries. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*. 10.

P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. 1999. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems* 14, 3 (May 1999), 54–62.

A. V. Papadopoulos, M. Maggio, F. Terraneo, and A. Leva. 2015. A dynamic modelling framework for control-based computing system design. *Mathematical and Computer Modelling of Dynamical Systems* 21, 3 (2015), 251–271.

A. V. Papadopoulos, A. Ali-Eldin, K.-E. Årzén, J. Tordsson, and E. Elmroth. 2016. PEAS: A performance evaluation framework for auto-scaling strategies in cloud applications. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 1, 4, Article 15 (2016).

S. Parekh. 2010. *Feedback Control Techniques for Performance Management of Computing Systems*. Ph.D. Dissertation. Seattle, WA. Advisor(s): Lazowska, Edward D.

S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. 2002. Using control theory to achieve service level objectives in performance management. *Real-Time Systems* 23, 1/2 (2002), 15.

T. Patikirikorala, A. Colman, J. Han, and L. Wang. 2012. A systematic survey on the design of self-adaptive software systems using control engineering approaches. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'12)*. IEEE Press, Piscataway, NJ, 33–42.

N. Piterman, A. Pnueli, and Y. Sa'ar. 2006. Synthesis of reactive(1) designs. In *Verification, Model Checking, and Abstract Interpretation*. Lecture Notes in Computer Science, Vol. 3855.

A. Pnueli. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*.

A. Pnueli and R. Rosner. 1989. On the synthesis of a reactive module. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*.

A. Puggelli, W. Li, A. L. Sangiovanni-Vincentelli, and S. A. Seshia. 2013. Polynomial-time verification of PCTL properties of MDPs with convex uncertainties. In *Computer Aided Verification*. Springer.

M. L. Puterman. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, New York, NY.

W. Qian, X. Peng, B. Chen, J. Mylopoulos, H. Wang, and W. Zhao. 2014. Rationalism with a dose of empiricism: Case-based reasoning for requirements-driven self-adaptation. In *2014 IEEE 22nd International Requirements Engineering Conference (RE'14)*. 113–122.

R. R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. 2010. Cyber-physical systems: The next computing revolution. In *Proceedings of the 47th Design Automation Conference*.

P. J. G. Ramadge and W. M. Wonham. 1989. The control of discrete event systems. *Proceedings of the IEEE* 77, 1 (Jan. 1989), 81–98.

A. J. Ramirez, B. H. C. Cheng, N. Bencomo, and P. Sawyer. 2012a. *Relaxing Claims: Coping with Uncertainty While Evaluating Assumptions at Run Time*. Springer, Berlin, 53–69.

A. J. Ramirez, A. C. Jensen, and B. H. C. Cheng. 2012b. A taxonomy of uncertainty for dynamically adaptive systems. In *2012 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'12)*. IEEE, 99–108.

J. Rao, Y. Wei, J. Gong, and C.-Z. Xu. 2011. DynaQoS: Model-free self-tuning fuzzy control of virtualized resources for QoS provisioning. In *International Workshop on Quality of Service (IWQoS'11)*. IEEE, 1–9.

T. Rompf and M. Odersky. 2010. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering (GPCE'10)*. 127–136.

M. Salehie and L. Tahvildari. 2009. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems* 4, 2, Article 14 (May 2009), 14:1–14:42 pages.

M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. 2014. Paraprox: Pattern-based approximation for data parallel applications. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 35–50.

R. Sanz and J. Zalewski. 2003. Pattern-based control systems engineering. *IEEE Control Systems* 23, 3 (June 2003), 43–60.

R. Scattolini. 2009. Architectures for distributed and hierarchical model predictive control–a review. *Journal of Process Control* 19, 5 (2009), 723–731.

S. Skogestad and I. Postlethwaite. 2007. *Multivariable Feedback Control: Analysis and Design*. Vol. 2. Wiley.

V. E. S. Souza, A. Lapouchnian, and J. Mylopoulos. 2011. System identification for adaptive software systems: A requirements engineering perspective. In *Conceptual Modeling (ER'11)*. Vol. 6998.

G. Stein. 2003. Respect the unstable. *IEEE Control Systems* 23, 4 (Aug. 2003), 12–25.

Q. Sun, G. Dai, and W. Pan. 2008. LPV model and its application in web server performance control. In *2008 International Conference on Computer Science and Software Engineering*, Vol. 3. 486–489.

P. Tabuada. 2007. Event-triggered real-time scheduling of stabilizing control tasks. *IEEE Transactions on Automatic Control* 52, 9 (Sept. 2007), 1680–1685.

G. Tesauro. 2007. Reinforcement learning in autonomic computing: A manifesto and case studies. *IEEE Internet Computing* 11, 1 (2007), 22–30.

M. Tribastone. 2013. A fluid model for layered queueing networks. *IEEE Transactions on Software Engineering* 39, 6 (2013), 744–756.

N. M. Villegas, H. A. Müller, G. Tamura, L. Duchien, and R. Casallas. 2011. A framework for evaluating quality-driven self-adaptive software systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'11)*.

T. Vogel and H. Giese. 2014. Model-driven engineering of self-adaptive software with EUREMA. *ACM Transactions on Autonomous and Adaptive Systems* 8, 4, Article 18 (2014), 33 pages.

V. Vyatkin. 2013. Software engineering in industrial automation: State-of-the-art review. *IEEE Transactions on Industrial Informatics* 9, 3 (Aug. 2013), 1234–1249.

L. Wang, J. Xu, and M. Zhao. 2015. QoS-driven cloud resource management through fuzzy model predictive control. In *International Conference on Autonomic Computing (ICAC'15)*. IEEE.

W.-P. Wang, D. Tipper, and S. Banerjee. 1996. A simple approximation for modeling nonstationary queues. In *Proceedings of the IEEE 15th Annual Joint Conference of the IEEE Computer Societies, Networking the Next Generation (INFOCOM'96).* Vol. 1. 255–262.

J. A. Whittaker and J. H. Poore. 1993. Markov analysis of software specifications. *ACM Transactions on Software Engineering Methodologies* 2, 1 (Jan. 1993), 93–106.

B. Wittenmark, K. Åström, and K.-E. Årzén. 2002. *Computer Control: An Overview*. Technical Report.

J. Xu, M. Zhao, J. Fortes, R. Carpenter, and M. Yousif. 2007. On the use of fuzzy modeling in virtualized data center management. In *4th International Conference on Autonomic Computing*.

R. Yager and L. Zadeh. 2012. *An Introduction to Fuzzy Logic Applications in Intelligent Systems*. Springer.

T. C. Yang. 2006. Networked control system: A brief survey. *IEE Proceedings - Control Theory and Applications* 153, 4 (July 2006), 403–412.

N. Zhan, S. Wang, and H. Zhao. 2013. Formal modelling, analysis and verification of hybrid systems. In *Unifying Theories of Programming and Formal Engineering Methods: International Training School on Software Engineering*. Springer, 207–281.

M. Zhang, B. Selic, S. Ali, T. Yue, O. Okariz, and R. Norgren. 2016. Understanding Uncertainty in Cyber-Physical Systems: A Conceptual Model. Technical Report. Simula Research Laboratory, 2015-3.

K. Zhou, J. C. Doyle, and K. Glover. 1996. *Robust and Optimal Control*. Prentice-Hall, Upper Saddle River, NJ.

X. Zhu, M. Uysal, Z. Wang, S. Singhal, A. Merchant, P. Padala, and K. Shin. 2009. What does control theory bring to systems research? *SIGOPS Opererating System Review* 43 (2009), 62–69.

P. Zuliani, A. Platzer, and E. M. Clarke. 2010. Bayesian statistical model checking with application to simulink/stateflow verification. In *Proceedings of the 13th ACM International Conference on Hybrid Systems: Computation and Control (HSCC'10)*. ACM, New York, NY, 243–252.

Received September 2015; revised October 2016; accepted November 2016