Towards Bridging the Gap between Control and Self-Adaptive System Properties

Javier Cámara
University of York, UK
javier.camaramoreno@york.ac.uk

Alessandro V. Papadopoulos
Mälardalen University, Sweden
alessandro.papadopoulos@mdh.se

Thomas Vogel
Humboldt University Berlin, Germany
thomas.vogel@cs.hu-berlin.de

Danny Weyns
KU Leuven, Belgium; Linnaeus, Sweden
danny.weyns@kuleuven.be

David Garlan
Carnegie Mellon University, USA
garlan@cs.cmu.edu

Shihong Huang
Florida Atlantic University, USA
shihong@fau.edu

Kenji Tei
Waseda University, Japan
ktei@aoni.waseda.jp

ABSTRACT

Two of the main paradigms used to build adaptive software employ different types of properties to capture relevant aspects of the system's run-time behavior. On the one hand, control systems consider properties that concern static aspects like stability, as well as dynamic properties that capture the transient evolution of variables such as settling time. On the other hand, self-adaptive systems consider mostly non-functional properties that capture concerns such as performance, reliability, and cost. In general, it is not easy to reconcile these two types of properties or identify under which conditions they constitute a good fit to provide run-time guarantees. There is a need of identifying the key properties in the areas of control and self-adaptation, as well as of characterizing and mapping them to better understand how they relate and possibly complement each other. In this paper, we take a first step to tackle this problem by: (1) identifying a set of key properties in control theory, (2) illustrating the formalization of some of these properties employing temporal logic languages commonly used to engineer self-adaptive software systems, and (3) illustrating how to map key properties that characterize self-adaptive software systems into control properties, leveraging their formalization in temporal logics. We illustrate the different steps of the mapping on an exemplar case in the cloud computing domain and conclude with identifying open challenges in the area.

CCS CONCEPTS

• **Software and its engineering → Extra-functional properties**; *Formal software verification.*

SEAMS '20, May 25–26, 2020, Seoul, South Korea

ACM ISBN AAA-B-CCC-XXXX-X/YY/ZZ…$XX.YY
https://doi.org/10.1145/1122445.1122456

KEYWORDS

self-adaptation, control theory, nonfunctional requirements

ACM Reference Format:
Javier Cámara, Alessandro V. Papadopoulos, Thomas Vogel, Danny Weyns, David Garlan, Shihong Huang, and Kenji Tei. 2020. Towards Bridging the Gap between Control and Self-Adaptive System Properties. In *SEAMS '20: 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, May 25–26, 2020, Seoul, South Korea.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Two of the main paradigms used to build adaptive software employ different types of properties to capture relevant aspects of the system's run-time behavior. On the one hand, control systems consider properties that concern static aspects like stability, as well as dynamic properties that capture transient aspects such as settling time. On the other hand, self-adaptive systems consider mostly non-functional properties that include concerns such as performance, cost, and reliability.

Self-adaptive software can clearly benefit from the potential that control theory provides in terms of enabling better analyzability and enforcement of constraints on run-time system behavior. Being able to formally reason about the non-functional concerns of a system (e.g., security, energy, performance) in terms of control properties in the presence of an unpredictable environment can optimize operation and improve the level of assurances that engineers can provide about the systems they build.

However, applying control theory to software systems poses a set of challenges that do not exist in other domains [9, 20]. One of the main challenges is that control-based solutions demand the availability of precise mathematical models that capture both the dynamics of the system under control, as well as the properties that engineers want to impose and reason about. When control is applied to physical plants, the laws that govern the system are captured by accurate mathematical models that are well-understood, and relevant properties like stability or performance are formally characterized by definitions that are precise and standard in the control community [4].

While obtaining accurate models of non-functional aspects of software behavior can to some extent be achieved using different methods like *system identification* [23], the self-adaptive software systems community still lacks a standard repertoire of run-time properties formally characterized in a way that makes them amenable to formal analysis using techniques applied by software engineers in self-adaptive systems (e.g., run-time verification, model checking). Having such a repertoire would not only help individual system designers express and check certain common fundamental properties, but also help promote norms for system assurance across the community of adaptive systems developers.

Solving in software the kind of problems that control theory solves in other domains entails understanding how control properties relate to software requirements and formally characterizing such properties in a way that facilitates their instantiation and automated analysis using standard tools.

To advance the understanding of how self-adaptive system requirements relate to control properties, in this paper we: (1) identify a set of key properties in control theory, (2) illustrate the formalization of some of these properties employing temporal logic languages commonly used to engineer self-adaptive software systems, and (3) indicate how to map key properties that characterize self-adaptive software systems into control properties, leveraging their formalization in temporal logics. We illustrate the different steps of the mapping on an exemplar case in the cloud computing domain and conclude with identifying open challenges in the area.

2 BACKGROUND

In this section, we first present a basic set of concepts in control systems, followed by a description of a general class of discrete abstractions which are employed to capture the non-functional behavior of self-adaptive systems at run time.

2.1 Control Terminology

In this paper, we focus mainly on continuous-time signals and systems, but equivalent definitions are present in the case of discrete-time [4].

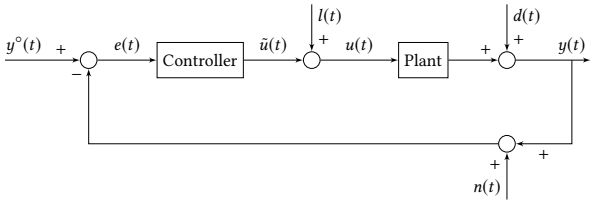

Figure 1: Control scheme.

First, consider the control scheme represented in Figure 1. The two main blocks represent the **Controller** and the **Plant** respectively. The Plant is the object that we want to control. Let $t \in \mathbb{R}$ be the continuous-time, where $\mathbb{R}$ indicates the real numbers; all the signals that are introduced next are functions of the time t. The m **inputs** of the plant are represented as $u(t) \in \mathbb{R}^m$, and in computing systems are typically referred as *control parameters*, or *tuning parameters*. The p **outputs** of the plant are typically represented as $y(t) \in \mathbb{R}^p$, and in computing systems are typically referred as *measurements* or *sensors data*.

For every output $y(t)$ of the plant, one defines a desired behavior for it, which in control terms is called a **setpoint** or **reference signal**, and is represented by $y^\circ(t) \in \mathbb{R}^p$.

The difference between the desired behavior and the actual behavior of the plant is called **error**, and is represented as $e(t) \in \mathbb{R}^p$:

$$e(t) = y^\circ(t) - y(t).$$

The controller is a decision-making mechanism that given the error, decides what is the value of the **control signal** $\tilde{u}(t) \in \mathbb{R}^m$ in order to make the error converge to zero. In principle, the control signal and the plant input should be the same, i.e., $\tilde{u}(t) = u(t)$, but in practice, there might be a **load disturbance** $l(t) \in \mathbb{R}^m$, that affects the controller decision. Therefore, it holds that

$$u(t) = \tilde{u}(t) + l(t).$$

The load disturbance is one of the main disturbances that affect the performance of control systems.

In addition, there might be a disturbance that is acting directly on the output of the plant, which is called **output disturbance**, and it is represented as $d(t) \in \mathbb{R}^p$. Finally, there is **noise** $n(t) \in \mathbb{R}^p$ that affects the measurements that one takes of the output. These two last sources of disturbances are typically "high-frequency" disturbances, and can be counteracted by a suitable filtering at design time of the controller.

As a main reference to these concepts, the interested reader can refer to the publicly available book by Åström and Murray [4].

2.2 Discrete Models

We *consider the self-adaptive system as a black-box on which a set of output variables can be monitored over time.* Concretely, we model the non-functional run-time behavior of a self-adaptive system as a transition system that captures the evolution over time of a set of relevant variables (i.e., state is characterized by a collection of n real-valued random variables $Y = \{y_1, \ldots, y_n\}$). *These variables can be considered to be analogous to the outputs $y(t)$ in a control system.* Sampling these variables in space and time results in their quantization and time discretization.

Let $[\alpha_i, \beta_i]$ be the range of y_i, with $\alpha_i, \beta_i \in \mathbb{R}$, and $\eta_i \in \mathbb{R}^+$ be its quantization parameter. Then, y_i takes its values in the set:

$$[\mathbb{R}]_{y_i} = \{r : \mathbb{R} \mid r = k\eta_i,\ k \in \mathbb{Z},\ \alpha_i \le r \le \beta_i\}.$$

Hence, given an observed value of y_i at time t (denoted as $y_i(t)$), the corresponding quantized value is obtained as:

$$quant(y_i(t)) = \min_{r \in [\mathbb{R}]_{y_i}}(\arg\min(|y_i(t) - r|)).$$

Variables in Y define a state-space $[\mathbb{R}^n]_Y = [\mathbb{R}]_{y_1} \times \ldots \times [\mathbb{R}]_{y_n}$. Furthermore, we assume a time discretization parameter $\tau \in \mathbb{R}^+$ associated with the sampling period established for the observation of variables, determining the transition time.

Figure 2 compares an arbitrary continuous system output $y(t)$ with its quantized counterpart $y_q(t)$[1] in the discrete timeline. $y_q(t)$ takes values only in multiples of η_y, and is represented in the figure as constant for intervals of duration τ.

Discrete models can be enriched with rewards and costs that help capture quantitative aspects of system behavior (e.g., elapsed time, energy consumption, cost) in a precise manner. These rewards can be employed as building blocks to reason about properties that capture quantitative aspects of system behavior over time.

[1] For convenience, we write in the following $y_q(t)$ instead of $quant(y(t))$.

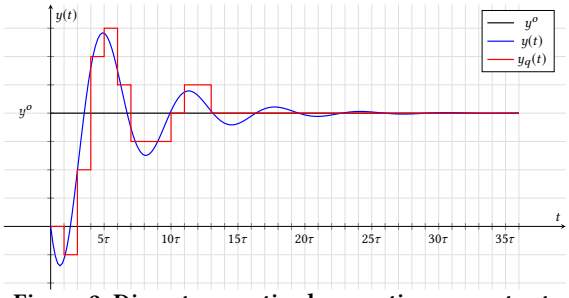

Figure 2: Discrete quantized vs continuous output.

A *reward structure* is a pair (ρ, ι), where $\iota : [\mathbb{R}^n]_Y \to \mathbb{R}_{\geq 0}$ is a function that assigns rewards to system states, and $\rho : [\mathbb{R}^n]_Y \times [\mathbb{R}^n]_Y \to \mathbb{R}_{\geq 0}$ is a function assigning rewards to transitions.

State reward $\iota(s)$ is acquired in state $s \in [\mathbb{R}^n]_Y$ per time step, that is, each time that the system spends one time step in s, the reward accrues $\iota(s)$. In contrast, $\rho(s, s')$ is the reward acquired every time that a transition between s and s' occurs.

For illustration purposes, we assume that rewards are defined as sets of pairs (pd, r), where pd is a predicate over states $[\mathbb{R}^n]_Y$, and $r \in \mathbb{R}_{\geq 0}$ is the accrued reward when $s \in [\mathbb{R}^n]_Y \models pd$. If the pair (pd, r) corresponds to a transition reward, the reward is accrued when a transition from a source state $s \in [\mathbb{R}^n]_Y \models pd$ occurs.

3 ILLUSTRATION EXEMPLAR: RUBIS

We illustrate our formalization of properties on RUBiS [1], an open-source application that implements the functionality of an auctions website. Figure 3 depicts the architecture of RUBiS, which consists of a web server tier that receives requests from clients using browsers, and a database tier that acts as a data provider for the web tier. The system also includes a load balancer to distribute requests among web servers using a round-robin policy. When a web server receives a page request from the load balancer, it accesses the database to obtain the data required to render dynamic page content. The only relevant property of the operating environment that we consider in this scenario is the request arrival rate prescribed by the workload induced on the system.

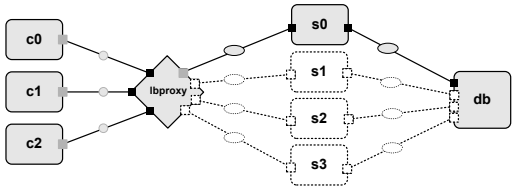

Figure 3: RUBiS architecture.

The system includes two actuation points that can be operationalized by a controller to make the system self-adaptive and deal with the changing request arrival rate:

• *Server Addition/Removal.* Server addition has an associated latency, whereas the latency for server removal is assumed to be negligible.
• *Dimmer.* The version of RUBiS used for our comparison follows the *brownout* paradigm [13], in which the response to a request includes mandatory content (e.g., the details of a product), and optional content such as recommendations of related products. A *dimmer* parameter (taking values in the interval $[0, 1]$) can be set to control the proportion of responses that include optional content. The goals of the target system are summarized in two functional and three non-functional requirements (Table 1).

Table 1: Requirements for RUBiS.

Functional Requirements	
R1	The target system shall respond to every request for serving its content.
R2	The target system shall serve optional content to the connected clients.
Non-Functional Requirements	
NFR1	The target system shall demonstrate high performance. The average response time r should not exceed T.
NFR2	The target system shall provide high availability of the optional content. Subject to NFR1, the percentage of requests with optional content (i.e., the dimmer value d) should be maximized.
NFR3	The target operating system shall operate under low cost. Subject to NFR1 and NFR2, the cost (i.e., the number of servers s) should be minimized.

There is a strict preference order among the non-functional requirements that deal with optimization, so trade-offs among different dimensions to be optimized are not possible (i.e., no solution should compromise maximizing the percentage of requests with optional content to reduce cost). The imposition of a preference order is aimed at better capturing real scenarios and is not a limitation imposed by any of the compared approaches, which are also able to capture non-strict preference orders among requirements.

4 CHARACTERIZING CONTROL PROPERTIES

Control systems are usually concerned about four main objectives [9], namely: (a) *setpoint tracking*, which is related to achieving the specified setpoint whenever it is reachable, (b) *transient behavior*, concerned about how setpoints are reached, in particular in the presence of abrupt changes, (c) *robustness to inaccurate or delayed measurements*, related to the ability of a controller to behave correctly even when transient errors or delayed data is provided to it, and (d) *disturbance rejection*, related to the ability of avoiding any effect of external interferences on system goals. These high level objectives can be mapped in control theory into the satisfaction *by design* of properties like *stability*, *guaranteed settling time*, *integrated squared error*, that relate to the achievable runtime performance of the control system. In this section, we describe these properties, going from their mathematical formulation into their characterization in temporal logics commonly used in formal verification like LTL [16], CTL [6], and PCTL [10]. Other properties exist in control theory, but having a complete catalogue here is beyond the scope of this paper, and it is left as future work.

4.1 Stability

The concept of *stability* in control theory differs from the notion of stability used in self-adaptive software. A control system is stable even if the error $e(t)$ is not converging to zero, but it is bounded. More specifically, in control terms, if the initial value of system output $y(0)$ is "close" to the equilibrium value y°, then the evolution over time of the output $y(t)$ will be bounded (and not diverge) from y°. More formally:

$$stby \equiv \forall \epsilon > 0 \; \exists \delta(\epsilon) \mid \|y(0) - y^\circ\| < \delta(\epsilon) \Rightarrow \|y(t) - y^\circ\| < \epsilon, \forall t > 0 \quad (1)$$

A system is *asymptotically stable*, if it is stable (as per the previous definition), and also if the evolution over time of the system output

will eventually converge to y°. More formally:

$$as\_stby \equiv stby \wedge \lim_{t\to\infty} \|y(t) - y^\circ\| = 0 \quad (2)$$

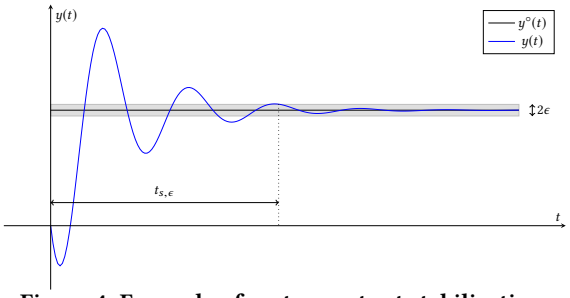

Figure 4: Example of system output stabilization.

Figure 4 shows the response of a system that eventually stabilizes within an error band (gray box) of width 2ϵ.

Characterization in Temporal Logic. Characterizing stability in temporal logic requires capturing the constraints imposed by the definition of stability given in Expression 1. Such characterization can be given on a quantized version of the variables and constants required to define stability:

$$[stby] \equiv \|y_q - y_q^\circ\| < \delta_q \Rightarrow \Box(\|y_q - y_q^\circ\| < \epsilon_q) \quad (3)$$

In Expression 3, the subscript q indicates that the constant or variable on which it appears is the quantized version of its continuous counterpart (i.e., $y_q(t) \equiv quant(y(t))$, cf. Section 2.2). It is worth noticing that variables in a software system are quantized by definition, and requiring a notion of asymptotic stability may be too restrictive. The current definition captures the same concept with ϵ_q being the resolution of the quantization or a tolerance parameter. Moreover, the absence of explicit time indexes is consistent with the implicit notion of time introduced by the temporal operators. For instance, when y_q is not within the scope of any temporal operator (like in the antecedent of the implication given in the formula), the expression refers to the value of the variable in the first state of the trace (i.e., $y_q \equiv y_q(0)$). However, if the same term is within the scope of a temporal operator as it happens with the $\Box$ on the right-hand side of the expression, then the same y_q refers to the value of $y_q(t)$ in all subsequent states of the discrete temporal line (i.e., $y_q(t)$ when $t = 0, \tau, 2\tau, \ldots$). The non-probabilistic version of this property is directly expressible in LTL and CTL (as $\mathbf{A}[stby]$), whereas its probabilistic version can employ the probability quantifier of PCTL (e.g., $P_{=?}[stby]$, $P_{\leq b}[stby]$).

Instantiation in RUBiS. Expression 4 instantiates $[stby]$, in a straightforward manner for the response time variable r, assuming a setpoint equivalent to the threshold T. It states that when the error becomes smaller than δ_q^r it will stay within the band $[T-\epsilon_q^r, T+\epsilon_q^r]$.

$$\|r_q - T\| < \delta_q^r \Rightarrow \Box(\|r_q - T\| < \epsilon_q^r) \quad (4)$$

4.2 Settling Time

One of the key indicators of how the system reaches its goals is *settling time* t_s, which is the time needed by the system to reach a new steady-state equilibrium.

For an arbitrary $\epsilon \in \mathbb{R}^+$, the *$\epsilon$-settling time* is defined by:

$$t_{s,\epsilon} \equiv \inf\{\delta \text{ s.t. } \|y(t) - y^\circ\| < \epsilon, \forall t \in [\delta, \infty]\} \quad (5)$$

In Expression 5, the settling time is captured as the infimum of the set of time values in the continuous timeline for which the error is bounded by ϵ in the following. Note that the infimum is the greatest lowest bound that always exists, meaning that it takes the value ∞ if the stability condition is never satisfied.

Characterization in Temporal Logic. In contrast with stability, which is a boolean property that is either satisfied by the system or not (Expression 3), settling time is a quantitative property and therefore we characterize it as a temporal logic expression that employs a reward quantifier. Since in this case the reward captures time, we assume the existence of a transition reward function $[\text{time}] \equiv (true, \tau)$ that accrues the time quantum employed for time in the discrete model whenever a transition in the discrete timeline is taken:

$$[t_{s,\epsilon}] \equiv \mathrm{R}^{[\text{time}]}_{=?}[\Diamond\Box\|y_q - y_q^\circ\| < \epsilon_q] \quad (6)$$

Expression 6 characterizes the settling time as the time reward accrued until the system reaches a state from which the error is bounded by ϵ_q. There are two aspects of this characterization that are important to highlight. First, the reachability formula accrues reward until it reaches a state that satisfies the reachability predicate, but the reward in the latter state is not included. Second, when the reachability predicate is not satisfied, the semantics of the reward quantifier assign an infinite reward as the value that is obtained when the expression is quantified (e.g., in PCTL, co-safe LTL with rewards). These two aspects make this characterization consistent with the definition given in Expression 5, which defines the settling time as the time instant immediately prior to the one in which the error is already bound by ϵ, and becomes infinite if the error is not always bound by ϵ, starting at some arbitrary point in the timeline. Note that, due to the nesting of temporal operators including $\Box$, this property is not (currently) directly expressible in temporal logics with Markovian rewards as implemented in probabilistic model checkers like PRISM [14] and Storm [7]. However, assuming finite traces in our discrete models, we can perform a preprocessing step on the traces, labeling explicitly states from which $\Box\|y_q - y_q^\circ\| < \epsilon_q$ as p, and then model check the property as:

$$\mathrm{R}^{[\text{time}]}_{=?}[\Diamond p] \quad (7)$$

Instantiation in RUBiS. Expression 8 instantiates $[t_{s,\epsilon}]$ with similar assumptions to those adopted for Expression 4.

$$\mathrm{R}^{[\text{time}]}_{=?}[\Diamond\Box\|r_q - T\| < \epsilon_q^r] \quad (8)$$

4.3 Integrated Squared Error

Relevant quantitative measures of a system's performance are also often based on the behavior of the error $e(t)$. We consider here as a representative index the *integrated squared of the error* (ISE):

$$\text{ISE} \equiv \int_0^T e^2(t)\mathrm{d}t \quad (9)$$

The ISE integrates the square of the error over time (see Figure 5), penalizing large errors more than smaller ones (the square of a large error will be much bigger). Control systems specified to minimize ISE of the tracking error $e(t)$, e.g., MPC or LQG [**?**], tend to eliminate large errors quickly, but tolerate small ones persisting for a long period of time. This often leads to fast responses, but with considerably low-amplitude oscillation.

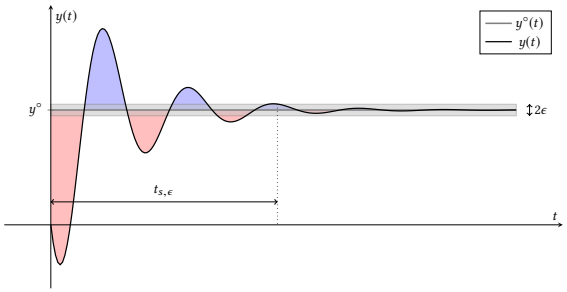

Figure 5: Illustration of the integrated squared error.

Characterization in Temporal Logic. Similar to the settling time, the ISE is a quantitative property and therefore we characterize it making use of a reward quantifier. Since in this case the reward has to capture accrued error over time, we assume the existence of a transition reward function $[\text{error}] \equiv (true, (\|y_q - y_q^\circ\|)^2)$ that accrues the square of the instantaneous error whenever a transition in the discrete temporal line is taken.

Then, we can write an expression that accrues the error reward over the discrete timeline before stability is achieved:

$$[ISE] \equiv \mathsf{R}^{[\text{error}]}_{=?}[\Diamond\Box\|y_q - y_q^\circ\| < \epsilon_q] \tag{10}$$

Due to the nesting of $\Diamond\Box$, this property is not directly expressible in PCTL/Co-safe LTL with rewards. However, under the same assumptions described for the settling time property, a similar model preprocessing step can enable its practical verification through a simpler probabilistic reachability property (cf. Expression 7).

Instantiation in RUBiS. We assume that RUBiS is working on steady state, but suddenly receives a spike on request arrival rate, causing the average response time r to go above threshold T (Figure 6). After violating the threshold, the system adds a server to drive down the response time below T. Before the system stabilizes, its response time may experience some oscillations that make r go above and below T several times. For simplicity, we assume $y^\circ = T$.

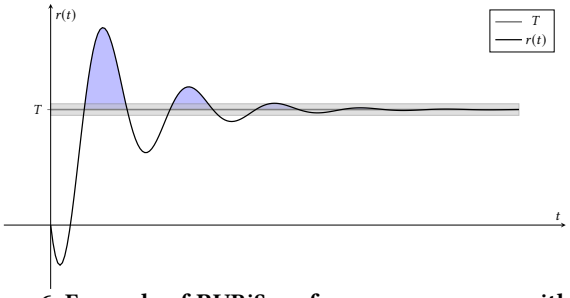

Figure 6: Example of RUBiS performance response with accrued positive squared error.

To obtain an indication of how well the system is adapting, we can employ a modified version of the $[ISE]$ property (Expression 10). In this case, we are only interested in accruing a penalty whenever the output of the system is above the threshold T, therefore we adapt the reward structure for the error, constraining it to accrue reward only whenever $r > T$, i.e., $[\text{penalty}] \equiv (r > T, (r - T)^2)$:

$$\mathsf{R}^{[\text{penalty}]}_{=?}[\Diamond\Box\|r - T\| < \epsilon_q] \tag{11}$$

We can observe that the accrued error corresponds to the colored areas enclosed by T and $r(t)$ in Figure 6. Since negative error (i.e., when $r < T$) does not constitute a violation of the response time threshold, we do not accrue it, in contrast with the more general property described in Expressions 9 and 10.

5 FORMALIZING NON-FUNCTIONAL REQUIREMENTS

The non-functional run-time behavior of self-adaptive systems can be captured by an external observer as a set of quantitative indicators that represent attributes of different concerns such as performance, cost, or availability. In this section, we employ the characterization of control properties in temporal logic introduced in the previous section as building blocks to formalize non-functional requirements in RUBiS.

NFR1. *The target system shall demonstrate high performance. The average response time r should not exceed T.* This requirement can be captured by combining temporal logic properties of: (i) stability as described by Expression 4, (ii) settling time as captured by Expression 8, and (iii) an integrated error property analogous to Expression 11 using the penalty, $[\text{penalty}] \equiv (r > T, r - T)$ which should be guaranteed to be always less or equal to zero, i.e.:

$$\mathsf{R}^{[\text{penalty}]}_{=?}[\Diamond\Box\|r - y_r^\circ\| < \epsilon_q] \leq 0 \tag{12}$$

Note that in the expression above, the error term $\|r - y_r^\circ\|$ does not make the simplifying assumption included in Expression 11, and incorporates an arbitrary setpoint different from T. This makes sense in a realistic setting because, if $y_r^\circ = T$, oscillations around the setpoint during transients would always result in response time threshold violations. This is also applicable to properties (i) and (ii) for this requirement.

NFR2.*The target system shall provide high availability of the optional content. Subject to NFR1, the percentage of requests with optional content (i.e., the dimmer value d) should be maximized.* Capturing this requirement requires instantiating the integrated error property on variable d, which should be always as close as possible to 1 (maximum optional content):

$$\mathsf{R}^{[\text{optional}]}_{=?}[\Diamond\Box\|1 - d_q\| < \epsilon^d_q] \tag{13}$$

where $[\text{optional}] \equiv (true, d)$. Note that in this case, the magnitude of the error is always below 1, so minimizing the non-squared error is a more sensible choice.

NFR3. *The target operating system shall operate under low cost. Subject to NFR1 and NFR2, the cost (i.e., the number of servers s) should be minimized.* The formalization of this requirement can be captured using the following properties defined over the response time variable r: (i) stability as described by Expression 4, (ii) settling time as captured by Expression 8. Finally, we can capture the penalty of using extra servers during the transient by employing an integral error property which should minimize the use of servers according to $[\text{penalty}] \equiv (r > T, s^2)$:

$$\mathsf{R}^{[\text{penalty}]}_{=?}[\Diamond\Box\|r_q - T\| < \epsilon^r_q] \tag{14}$$

Note that in this case, stability and settling time properties are defined over response time r, whereas penalty is defined over the number of servers employed s, making an interesting case in which formalizing a single requirement involves combining different control properties across variables.

All variables might present similar patterns in terms of control properties, but the composition of the self-adaptive properties is

non-trivial and might be realized in different ways. As a consequence, there is a need to incorporate high-level compositional operators to enable joint evaluation of the requirements. Alternatively, we might want to express priorities in how specific properties should be achieved.

6 RELATED WORK

We have grouped related work in four parts: control applied to computing systems, automatically generated control solutions, verification of control properties, and evaluation of quality properties.

Control Applied to Computing System. In 2004, Hellerstein et al. wrote a pioneering book on applying control theory to computing systems [11]. Over the years, control-based approaches have been applied extensively to computing systems, mostly focussing on controlling lower-level resources. Abdelzaher et al. apply different types of controller models (e.g., PI and PID) to deal with performance requirements of servers [2]. Wang et al. present DEUCON that allocates local controllers to computing units that only coordinate with neighbors [25]. Stability analysis is based on the location of poles of the composite system's transfer function. Imes et al. present CoPPer, a control-theoretic approach that applies adaptive control to meet soft performance goals by manipulating hardware power limits [12]. In contrast, our work targets a mapping between classic control properties and typical software qualities.

Automatically Generated Control Solutions. To deal with the complexity of control theory, researchers have started investigating automatic generation of control solutions to adapt software [9, 22, 26]. Filieri et al. introduce the push-button methodology (PBM) that automatically constructs a linear model of a software system for a PI controller to adapt the system for one setpoint goal [8]. Shevtsov et al. propose a solution to control a software system for multiple goals, including an optimization goal [21]. Maggio et al. apply model-predictive control (MPC) to software adaptation [15], while Anagelopoulos et al. apply a requirements-driven approach with MPC [3]. These approaches highlight properties that are important from a control-theoretic viewpoint, but this accounts for only one side of the problem we target in this paper, namely, a rigorous specification and verification of classic control properties.

Verification of Control Properties. Some work exits on the formalisation and verification of properties of control systems. We highlight two representative examples. Preuse and Hanisch apply model checking to verify safety, liveliness and deadlock properties of manufacturing control systems that are specified in temporal logic [17]. Yan et al. use approximate bisimulation for comparing the similarity between a complex (continuous) cyber-physical system and a (discretized) higher level model of it [28]. The authors illustrate the approach for a safety property. Our work complements these approaches by focusing on typical software quality properties and the formal mapping of these with control properties.

Evaluation of Quality Properties. A number of approaches zoom in on the evaluation of quality properties in self-adaptive systems. Weyns and Ahmad [27] performed a systematic literature review identifying the main quality properties considered in self-adaptation: efficiency/performance of the system (55% of the studies), reliability (41%), and flexibility (28%). Reinecke et al. [19] propose a payoff metric to measure the "success" of adaptation. This metric is a user-defined function aggregating QoS metrics observed on the running system similarly to a utility function. Villegas et al. [24] present a framework to evaluate adaptation properties, i.e., stability, accuracy, settling time, overshoot, robustness, termination of adaptation, consistency, scalability, and security. The properties are informally defined and mapped to software qualities based on examples from literature. Raibulet et al. [18] focus on quality attributes to evaluate the utility of a self-adaptive system, and software metrics to evaluate the quality of the adaptation at runtime, whereas Cámara and de Lemos [5] evaluate resilience properties formalized in PCTL. Each of these approaches contributes to a better understanding of quality properties and their evaluation from a software engineering point of view. However, this only accounts for one side of the mapping problem we target in this paper, i.e., a traditional software engineering perspective.

Conclusion. While control theory and self-adaptive systems contribute knowledge about properties in their domain, there is little understanding on the mapping between the two types of properties, which is precisely the target of the research presented in this paper.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we have taken the first step in bridging the gap between control and self-adaptive system properties. We have (1) identified key properties in control theory (stability, settling time, and integral error), (2) formalized these properties in temporal logic languages, which are typically used to specify properties (requirements) of software systems, and (3) illustrated how non-functional properties of self-adaptive systems (performance, availability, and costs) can be mapped into these control properties by using this formalization and the RUBiS exemplar. To achieve the formalization and mapping, we have discussed the abstraction of transition systems describing discrete state spaces on which self-adaptive system attributes are measured and how this abstraction is able to represent continuous system dynamics in which control properties are typically characterized. Models of such transition systems and control properties formalized in a temporal logic can serve as input for off-the-shelf run-time verification tools and model checkers.

This approach advances the understanding of how non-functional requirements relate to control properties (e.g., which requirements can be characterized by which control properties) and paves the way for an improved operation and assurance of self-adaptive systems via formal reasoning (e.g., by run-time verification) based on control. Our approach is currently limited by the set of control properties that we have formalized, requirements that we have mapped into control properties (cf. previous paragraph), and the expresiveness of temporal logics, which might not be able to fully capture the nuances of some control properties (cf. Section 4.2).

Our long-term goal is to understand whether control theory can be used as a formal foundation for specifying and analyzing self-adaptive systems, and if so, under which conditions. Towards that goal, work is needed to identify further corresponding and complementing properties between self-adaptive and control systems (e.g., whether real-time or security requirements can be mapped into control properties), and to leverage the formalization of properties for a formal assessment of controllers in self-adaptive systems (e.g., to provide guarantees for the correctness of controllers).

REFERENCES

[1] [n.d.]. Rice University Bidding System. ([n. d.]). http://rubis.ow2.org.

[2] T. F. Abdelzaher, J. A. Stankovic, Chenyang Lu, Ronghua Zhang, and Ying Lu. 2003. Feedback performance control in software services. *IEEE Control Systems Magazine* 23, 3 (June 2003), 74–90. https://doi.org/10.1109/MCS.2003.1200252

[3] K. Angelopoulos, A. V. Papadopoulos, V. Silva Souza, and J. Mylopoulos. 2018. Engineering Self-Adaptive Software Systems: From Requirements to Model Predictive Control. *ACM Transactions on Autonomous and Adaptive Systems* 13, 1, Article 1 (April 2018), 27 pages. https://doi.org/10.1145/3105748

[4] K.J. Åström and R.M. Murray. 2010. *Feedback Systems: An Introduction for Scientists and Engineers*. Princeton University Press. http://www.cds.caltech.edu/~murray/amwiki/index.php/Main_Page

[5] Javier Cámara and Rogério de Lemos. 2012. Evaluation of resilience in self-adaptive systems using probabilistic model-checking. In *7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2012, Zurich, Switzerland, June 4-5, 2012*, Hausi A. Müller and Luciano Baresi (Eds.). IEEE Computer Society, 53–62.

[6] E. Clarke and E Emerson. 1981. Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In *Logics of Programs, Workshop, Yorktown Heights, New York, USA, May 1981 (Lecture Notes in Computer Science)*, D. Kozen (Ed.), Vol. 131. Springer, 52–71.

[7] C. Dehnert, S. Junges, J.P. Katoen, and M. Volk. 2017. A Storm is Coming: A Modern Probabilistic Model Checker. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II (Lecture Notes in Computer Science)*, R. Majumdar and V. Kuncak (Eds.), Vol. 10427. Springer, 592–600.

[8] A. Filieri, C. Ghezzi, and G. Tamburrelli. 2012. A Formal Approach to Adaptive Software: Continuous Assurance of Non-functional Requirements. *Form. Asp. Comput.* 24, 2 (March 2012), 163–186. https://doi.org/10.1007/s00165-011-0207-2

[9] 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. V. Papadopoulos, S. Ray, A. Sharifloo, S. Shevtsov, M. Ujma, and T. Vogel. 2017. Control Strategies for Self-Adaptive Software Systems. *ACM Transactions on Autonomous and Adaptive Systems* 11, 4, Article 24 (Feb. 2017), 31 pages. https://doi.org/10.1145/3024188

[10] H. Hansson and B. Jonsson. 1994. A Logic for Reasoning about Time and Reliability. *Formal Asp. Comput.* 6, 5 (1994), 512–535.

[11] J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury. 2004. *Feedback Control of Computing Systems*. John Wiley Sons, Inc., USA.

[12] C. Imes, H. Zhang, K. Zhao, and H. Hoffmann. 2019. CoPPer: Soft Real-Time Application Performance Using Hardware Power Capping. In *2019 IEEE International Conference on Autonomic Computing (ICAC)*. 31–41. https://doi.org/10.1109/ICAC.2019.00015

[13] C. Klein, M. Maggio, K.E. Årzén, and F. Hernández-Rodriguez. 2014. Brownout: building more robust cloud applications. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. 700–711.

[14] M. Kwiatkowska, G. Norman, and D. Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *Computer Aided Verification - 23rd International Conference, CAV*, Vol. 6806. Springer, 585–591.

[15] M. Maggio, A. V. Papadopoulos, A. Filieri, and H. Hoffmann. 2017. Automated Control of Multiple Software Goals Using Multiple Actuators. In *11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany). ACM, New York, NY, USA, 373–384. https://doi.org/10.1145/3106237.3106247

[16] A. Pnueli. 1977. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 46–57.

[17] S. Preuße and H. Hanisch. 2011. Verifying functional and non-functional properties of manufacturing control systems. In *2011 3rd International Workshop on Dependable Control of Discrete Systems*. 41–46. https://doi.org/10.1109/DCDS.2011.5970316

[18] C. Raibulet, F. Arcelli Fontana, R. Capilla, and C. Carrillo. 2017. Chapter 13 - An Overview on Quality Evaluation of Self-Adaptive Systems. In *Managing Trade-Offs in Adaptable Software Architectures*, Ivan Mistrik, Nour Ali, Rick Kazman, John Grundy, and Bradley Schmerl (Eds.). Morgan Kaufmann, Boston, 325–352. https://doi.org/10.1016/B978-0-12-802855-1.00013-7

[19] P. Reinecke, K. Wolter, and A. van Moorsel. 2010. Evaluating the Adaptivity of Computing Systems. *Perform. Eval.* 67, 8 (Aug. 2010), 676–693. https://doi.org/10.1016/j.peva.2009.12.001

[20] Stepan Shevtsov, Mihaly Berekmeri, Danny Weyns, and Martina Maggio. 2018. Control-Theoretical Software Adaptation: A Systematic Literature Review. *IEEE Trans. Softw. Eng.* 44, 8 (Aug. 2018), 784–810. https://doi.org/10.1109/TSE.2017.2704579

[21] S. Shevtsov and D. Weyns. 2016. Keep It SIMPLEX: Satisfying Multiple Goals with Guarantees in Control-based Self-adaptive Systems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. ACM, New York, NY, USA, 229–241. https://doi.org/10.1145/2950290.2950301

[22] S. Shevtsov, D. Weyns, and M. Maggio. 2019. Self-Adaptation of Software Using Automatically Generated Control-Theoretical Solutions. In *Engineering Adaptive Software Systems - Communications of NII Shonan Meetings*. 35–55.

[23] A. Simpkins. 2012. System Identification: Theory for the User, 2nd Edition (Ljung, L.; 1999) [On the Shelf]. *IEEE Robotics Automation Magazine* 19, 2 (2012), 95–96.

[24] N. Villegas, H. 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* (Waikiki, Honolulu, HI, USA) *(SEAMS '11)*. ACM, New York, NY, USA, 80–89. https://doi.org/10.1145/1988008.1988020

[25] X. Wang, D. Jia, C. Lu, and X. Koutsoukos. 2007. DEUCON: Decentralized End-to-End Utilization Control for Distributed Real-Time Systems. *IEEE Transactions on Parallel and Distributed Systems* 18, 7 (July 2007), 996–1009. https://doi.org/10.1109/TPDS.2007.1051

[26] D. Weyns. 2018. Software Engineering of Self-Adaptive Systems. In *Handbook of Software Engineering*, Richard Taylor, Kyo Chul Kang, and Sungdeok Cha (Eds.). Springer. https://lirias.kuleuven.be/handle/123456789/578653

[27] D. Weyns and T. Ahmad. 2013. Claims and Evidence for Architecture-Based Self-adaptation: A Systematic Literature Review. In *Software Architecture*, Khalil Drira (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 249–265.

[28] G. Yan, L. Jiao, Y. Li, S. Wang, and N. Zhan. 2016. Approximate Bisimulation and Discretization of Hybrid CSP. In *FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings*. 702–720. https://doi.org/10.1007/978-3-319-48989-6_43