Bridging the Gap between Control and Self-Adaptive System Properties: Identification, Characterization, and Mapping

Javier Cámara, David Garlan, Shihong Huang, Masako Kishida, Alberto Leva, Hiroyuki Nakagawa, Alessandro Papadopoulos, Yasuyuki Tahara, Kenji Tei, Thomas Vogel, and Danny Weyns

September 21, 2017

Contact: Javier Cámara *jcmoreno@cs.cmu.edu*

Abstract

Context: 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, cost, and reliability.

Problem: 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 guarantees about relevant aspects of the system at run-time. 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.

Method: (1) Identify key properties in the two areas, (2) express them rigorously in a common language, (3) map properties in the two areas, and (4) analyze commonalities, differences, and potential complementarities among the different properties. We will use a simple use case to illustrate all the steps.

Expected Results: Obtain a catalog of key properties in control and self-adaptive systems, a set of patterns for specification of (possibly a subset of) those properties in a temporal logic language, a mapping between properties in both areas, and some insights into how to better combine formal guarantees obtained from control and other approaches.

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 different concerns such as performance, cost, and reliability.

Self-adaptive software can clearly benefit from the potential that control theory provides in terms of enabling the analyzability of run-time system behavior. Being able to formally reason about the non-functional concerns of a system (e.g., security, energy, performance) in the presence of an oftentimes 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 [5]. 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 [3].

While obtaining accurate models of non-functional aspects of software behavior can to some extent be achieved using different methods like *system identification* [9], 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).

Solving in software the kind of problems that control 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 improve on the current situation, our goal is to develop a preliminary catalog of key properties in control and self-adaptive systems, a set of patterns for specification of those properties in a temporal logic language, a mapping between properties in both areas, and some insights into how to better combine formal guarantees obtained from control and other approaches.

In the remainder of this document, we first introduce in Section 2 some background on control and self-adaptive systems, as well as about the kind of discrete models employed to capture system behavior. Moreover, we also include a brief overview of the temporal logic language employed to formalize properties. Next, Section 3 presents an overview of RUBiS [1], a self-adaptive web multi-tiered system that we employ as a running example to illustrate properties. Section 4 presents an overview of key properties in the areas of control and self-adaptive systems, which are later expanded in Sections 5 and 6, respectively. Finally, Section 7 presents a roadmap that discusses directions for future work.

2 Preliminaries and terminology

In this section, we first present a basic set of concepts in control systems, followed by a general model of self-adaptive system. The remainder of the section presents the kind of discrete abstraction employed to capture the non-functional behavior of self-adaptive systems at run-time, as well as the formal language employed to characterize properties.

2.1 Control Concepts

This section is devoted to basic definitions and to the terminology used in this document. The focus will be mainly on continuous-time systems, but similar concepts can be found for discrete-time systems. As a main reference the interested reader is referred to the publicly available book [3].

First, consider the control scheme represented in Figure 1.

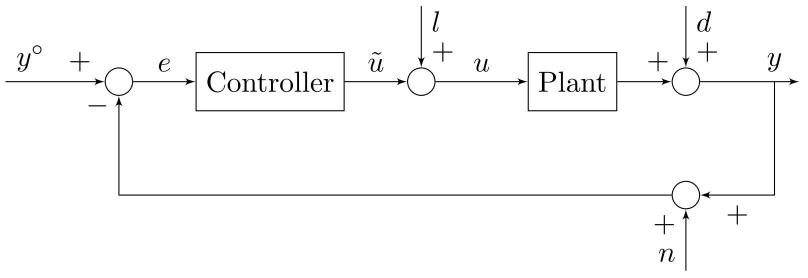


Figure 1: Control scheme.

The two main blocks represent the **Controller** and the **Plant** respectively. The Plant is the object that we want to control. The **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 **outputs** of the plant are typically represented as $y(t) \in \mathbb{R}^p$, and in computing systems are typically referred as *measurements*.

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

The difference between the desired behavior and the actual behavior of the plant is called **error**, and is typically 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.

Table 1 summarizes the mentioned quantities.

Name	**Description**
$u(t) \in \mathbb{R}^m$	Plant inputs
$y(t) \in \mathbb{R}^p$	Plant output
$y^\circ(t) \in \mathbb{R}^p$	Setpoint
$e(t) \in \mathbb{R}^p$	Error
$\tilde{u}(t) \in \mathbb{R}^m$	Control signal
$l(t) \in \mathbb{R}^m$	Load disturbance
$d(t) \in \mathbb{R}^p$	Output disturbance
$n(t) \in \mathbb{R}^p$	Noise

Table 1: Names and notation.

2.2 Self-Adaptive Systems

We consider the model of a self-adaptive system depicted in Figure 2. The **environment** consists of all non-controllable elements that determine the operating conditions of the system (*e.g.*, hardware, network, physical context, etc.).

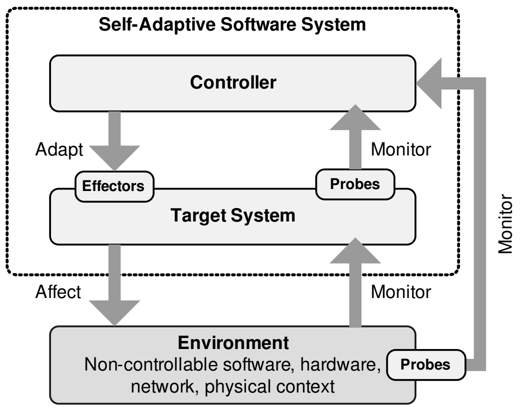

Self-Adaptive Software System
Controller
Adapt
Monitor
Effectors
Probes
Target System
Monitor
Affect
Monitor
Environment
Non-controllable software, hardware, network, physical context
Probes

Figure 2: Self-adaptive software system.

Regarding the system itself, we distinguish two main subsystems: a **target system** (or managed subsystem), which interacts with the environment by monitoring and affecting relevant variables associated with operating conditions, and a **controller** (or managing subsystem) that manages the target system, driving adaptation whenever it is required. Concretely, the controller carries out its function by: **(i)** monitoring the target system and environment through **probes** (or sensors) that provide information about the value of relevant variables, **(ii)** deciding whether the current state demands adaptation, and if this is the case, and **(iii)** applying a sequence of control actions through system-level **effectors** (or actuators).

Some of the key concerns with respect to the run-time behavior of self-adaptive systems are related to their non-functional attributes that include performance and cost, as well as attributes of dependability and resilience like availability, and reliability. Another major dimension of concern refers safety, that is, the absence of catastrophic consequences on the user and the environment, which can be caused by the self-adaptation [4].

2.3 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 time and space 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:

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

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

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

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.

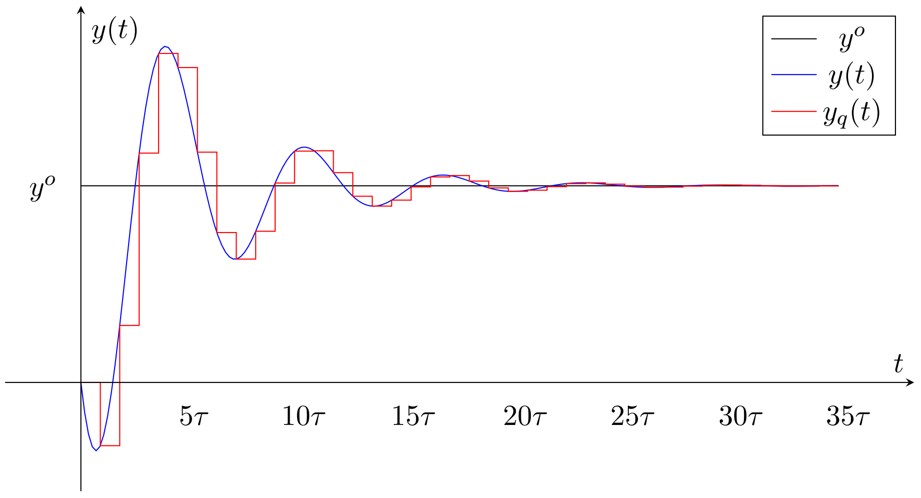


Figure 3: Example of discrete quantized vs continuous output.

Figure 3 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.[2]

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

[2]For illustration purposes, we represent the discretized system output with a large discretization parameter.

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 sophisticated properties that capture quantitative aspects of system behavior over time.

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.

2.4 Temporal Logic

Temporal logic is used to specify properties of transition systems, and in particular to claim something about the time at which a specific property holds. The term "time" here refers to the number of transitions that the transition system has taken so far, e.g., "after three steps." Note that in general, this notion of "time" can be related to statements about a discrete notion of real time by associating a fixed amount of time elapsed to every transition in the system (via the time discretization parameter τ described in Section 2.3).[3] For example, if we assume that τ=1ms, "after three milliseconds" would mean "after three transitions." Hence, we will see that temporal logic can be used in general to reason about the *ordering* of events in a system without introducing time explicitly (Section 2.4.1), although extensions can be employed to reason explicitly about quantitative aspects of systems that include time or probabilities (Section 2.4.2).

2.4.1 Linear Temporal Logic (LTL)

Linear Temporal Logic or *LTL* is used to make claims about a trace, considered as a sequence of states produced by a state machine describing a system.

Given a set of atomic propositions AP, any $ap \in AP$ is an LTL formula. Given two LTL formulas ϕ and ψ, then the following are also LTL formulas:

$$\begin{array}{c} \neg\phi \;\mid\; \phi \wedge \psi \;\mid\; \phi \vee \psi \;\mid\; \phi \implies \psi \;\mid\; \phi \Leftrightarrow \psi \;\mid \\ \Box\phi \;\mid\; \Diamond\phi \;\mid\; \bigcirc\,\phi \;\mid\; \phi \mathsf{U} \psi \end{array}$$

If we focus on the first of the two lines above, we can observe that it looks very much like propositional logic, including all its standard **logical operators**, which have exactly the same semantics here. The line in the bottom is the part of LTL that corresponds to its **temporal operators**, that is, operators that enable us to express properties about the ordering of the satisfaction of propositions in traces.

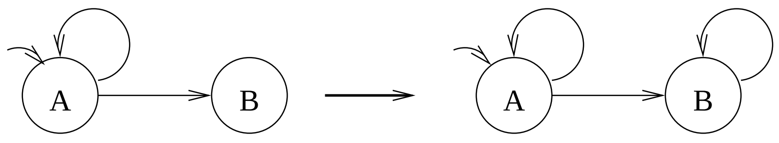
Informally, $\Box\phi$ states that ϕ will always hold in subsequent positions of the trace, $\Diamond\phi$ indicates that ϕ will eventually hold in a future position, $\bigcirc\phi$ states that ϕ holds in the next position, and $\phi\mathsf{U}\psi$ indicates that ψ holds in the current or a future position, and ϕ has to hold until that position (from that position onwards, ϕ does not necessarily have to hold).

In the variant of LTL that we employ in this document, we require sequences of states to be infinite, in order to simplify the interpretation of formulas.[4] However, the state machines that we

[3]Oftentimes, the amount of (real) time that a particular transition requires to occur is not fixed. For those cases, there are extended temporal logic languages that include clocks to represent real-time properties (e.g., Metric Linear Temporal Logic (MLTL) is an extension of LTL with clocks). We will not deal with such extensions in this document for the sake of simplicity.

[4]There are variants of LTL that are interpreted over finite traces. However, they introduce additional considerations that fall out of the scope of this document.

have presented so far can produce finite execution sequences, since we allow states that do not have any successors. We interpret these finite traces as infinite traces by simply repeating the last state of the sequence. Alternatively, you can think of this as a change to the state machine in which we add self-loops to all states that do not have a successor:



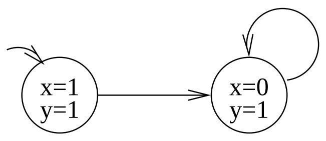
For example, the state machine above can produce the sequences:

1. A, A, A, ... (infinitely many A's),
2. A, ..., A, B, B, ... (finitely many A's, then infinitely many B's).

We denote the sequence by σ. Given such a sequence, we indicate that some property P holds for the i^{th} state in this sequence by writing:

$$(\sigma, i) \models P$$

The first state of the sequence is state 1. As an example, consider the following transition system with two variables x and y:



Consider the property $P = (x \geq y)$. The transition system in the example can produce only one sequence. In this sequence, P holds on the first (initial) state, but not on any later state. Thus, $(\sigma, 1) \models x \geq y$ is satisfied, but $(\sigma, 2) \models x \geq y$ is not.

We introduce the following shorthand: if we just write a property P without giving a state, we refer to the first state of the sequence, i.e., P is a shorthand for $(\sigma, 1) \models P$. If the state machine can produce more than one sequence, the claim is about the first state of all the sequences, i.e., about all the initial states.

In the remainder of this section, we will go over the main temporal operators of LTL, looking into their semantics and examples of how they are used.

The Box Operator. If we want to assert a safety property about a system, we can use the notation above in combination with a universal quantifier to specify that P holds in all the states of a sequence:

$$\forall j \in \mathbb{N} : (\sigma, j) \models P$$

LTL offers a convenient shorthand for this type of property: the *box operator*. It is applied as follows:

$$(\sigma, i) \models \Box P$$

The expression above asserts that P holds in all subsequent positions of a trace, starting in the i^{th} state, or formally:

$$\forall j \in \mathbb{N} | j \geq i : (\sigma, j) \models P$$

The box operator can be intuitively interpreted as "from now on, P holds". The box operator is sometimes written as "G", which stands for "Globally". As a shorthand, we write $\Box P$ for $(\sigma, 1) \models \Box P$ (i.e., $\Box P$ means that P is an invariant).

Example: Concurrent access to shared resources in multi-threaded programs can sometimes lead to erroneous program behavior. In order to avoid that, a common mechanism is to implement a critical section that will allow only one thread to access the shared resource at a time. Let us assume a program with two threads t_1 and t_2. Propositions cs_{t1} and cs_{t2} indicate that threads $t1$ and $t2$ are in the critical section, respectively. If we want to assert that the critical section is never accessed concurrently by more than one thread in this system, we can write the following invariant in LTL:

$$\Box\neg(cs_{t1} \wedge cs_{t2})$$

According to the semantics that defined for the box operator, this invariant can be interpreted as:

$$\forall i \in \mathbb{N} | (\sigma, i) \models \neg(cs_{t1} \wedge cs_{t2})$$

The Diamond Operator. If we want to specify a liveness property to assert whether some desirable condition P will eventually hold in our system, we can state the following:

$$\exists j \in \mathbb{N} : (\sigma, j) \models P$$

Again, LTL offers a shorthand for this kind of property: $(\sigma, i) \models \Diamond P$ denotes that there is a state in the sequence at or after the i^{th} position that satisfies P, or formally:

$$\exists j \in \mathbb{N} | j \geq i : (\sigma, j) \models P$$

This is called *diamond operator*, and can be informally interpreted as "eventually, P holds". The diamond operator is sometimes written as "F", which stands for "Finally".

Consider for instance the sequence $\sigma = (A, A, B, B, \ldots)$. For this sequence, $(\sigma, 1) \models \Diamond B$ is true, but $(\sigma, 3) \models \Diamond A$ is not.

Example: Suppose we want to claim that a program terminates. Let us denote a terminating state using the proposition *terminates*. We could say that the program eventually terminates by claiming that there exists some position j in the sequence in which *terminates* holds:

$$\exists j \in \mathbb{N} : (\sigma, j) \models \textit{terminates} \quad \text{or equivalently,} \quad \Diamond \textit{terminates}$$

2.4.2 Probabilistic Computation Tree Logic with Rewards (PRCTL)

Probabilistic Computation Tree Logic or *PCTL* [6] is employed in probabilistic model checking to quantify properties related to probabilities, as well as reward and costs in system specifications described using probabilistic state machines like discrete-time Markov chains (DTMC), Markov decision processes (MDP), or probabilistic timed automata (PTA).

In this document, we build on a version of PCTL extended with a reward quantifier targeted at checking properties over DTMCs extended with reward structures (PRCTL) [2]. Furthermore, we abstract away from probabilities and focus on the deterministic version of discrete transition systems, considering only the reward quantifier of PRCTL.

In the syntax definition below, Φ and ϕ are respectively, formulas interpreted over states and paths of a DTMC extended with rewards (D, ρ). Properties in PCTL are specified exclusively as state formulas. Path formulas have only an auxiliary role on probability and reward quantifiers P and R:

$$\Phi ::= \texttt{true} |\ a \mid \neg\Phi \mid \Phi \wedge \Phi \mid \mathsf{P}_{\sim pb}[\phi] \mid \mathsf{R}^r_{\sim rb}[\phi] \quad \phi ::= \bigcirc\Phi \mid \Phi \ \mathsf{U}\ \Phi,$$

where a is an atomic proposition, $\sim \in \{<, \leq, \geq, >\}, pb \in [0,1], rb \in \mathbb{R}^+_0$, and $r \in \rho$.

Intuitively, $\mathsf{P}_{\sim pb}[\phi]$ is satisfied in a state s of D if the probability of choosing a path starting in s that satisfies ϕ (denoted as $Pr_s(\phi)$ [5]) is within the range determined by $\sim pb$, where pb is a probability bound.

Quantification of properties based on $\mathsf{R}^r_{\sim rb}$ works analogously, but considering rewards, instead of probabilities. Concretely, an extended version of the reward operator $\mathsf{R}^{\mathsf{r}}_{=?}[\Diamond\ \phi]$ enables the quantification of the accrued reward r along paths that lead to states satisfying ϕ.

The intuitive meaning of path operators $\bigcirc$ and U is analogous to the ones in other standard temporal logics like LTL. Additional boolean and temporal operators are derived in the standard way (e.g., $\Diamond\Phi \equiv \texttt{true}\ \mathsf{U}\ \Phi$, $\Box\Phi \equiv \neg\Diamond\neg\Phi$).

[5]See [8] for details. In the following, we write $Pr_s(\phi)$ as $Pr(\phi)$ for simplicity.

3 Case Study

We illustrate our formalization of properties on the Rice University Bidding System (RUBiS) [1], an open-source application that implements the functionality of an auctions website. Figure 4 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. Our setup of RUBiS also includes a load balancer to support multiple servers in the web tier, which distributes requests among them following 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 the dynamic content of the page. The only relevant property of the operating environment that we consider in our adaptation scenario is the request arrival rate prescribed by the workload induced on the system.

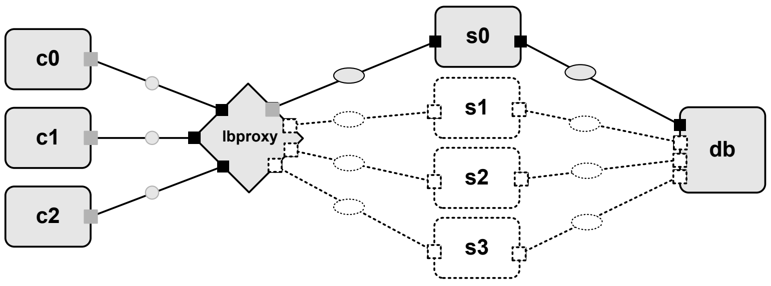


Figure 4: RUBiS architecture.

The system includes two actuation points that can be operationalized by a controller to make the system self-adaptive and deal with the changing loads induced by variations in the 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 [7], 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.

	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.

Table 2: Requirements for RUBiS.

The goals of the target system are summarized in two functional and three non-functional requirements (Table 2). 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 maximization of 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 Overview of Properties in Control and Self-Adaptive Systems

In this section we introduce a formalization of some key properties in control systems, and explore their potential use on self-adaptive systems. We start this exercise by identifying key properties in control (Table 3) that include:

- **Static properties**, which capture aspects about the steady-state (or the lack thereof) towards which the system evolves in the absence of external stimuli (e.g., stability, steady-state error).
- **Dynamic properties** that capture the transient aspects of system evolution before reaching the steady-state (e.g., oscillations, overshoot).

Control		**Self-Adaptive**
Static	**Dynamic**	
Stability	Settling Time	Performance
Asymptotic Stability	Oscillations	Cost
Steady state error	Overshoot	Reliability
...	Integrated Squared Error	Availability
	...	Security
		Resilience
		...

Table 3: Key properties in control and self-adaptive systems.

On the self-adaptive systems side, we can consider the quantitative attributes of the different non-functional concerns over time (e.g., response time for performance) as analogous to the outputs $y(t)$ in a continuous-time control system. However, it is worth considering that discretization of time will require averaging the measurement of values per time period, rather than considering their instantaneous value over the continuous timeline (e.g., average response time per τ-period).

We can employ the formalization of control properties with respect to self-adaptive system concerns to assess sophisticated properties about the system's run-time behavior (e.g., stability of system with respect to performance-response time).

5 Control Properties

Control systems are usually concerned about four main objectives [5]:

- *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 a given time threshold. In general, a self-adaptive system should be able to achieve the specified setpoint whenever it is reachable. Whenever the setpoint is not reachable, the controller should make sure that the measured value $y(t)$ is as close as possible to the desired value y^o.
- *Transient behavior.* Control theory is not only concerned about the fact that the setpoint is reached, but also about how this happens. The behavior of the system 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 y^o, 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 non-accurate 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 affect the closed-loop system other than the action of the controller. Disturbances should be 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.

These high level objectives have can be mapped into the "by design" satisfaction of the following properties:

- *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 (for example when the control variable is a certain number of virtual machines to be fired up for a specific software application).
- *Guaranteed settling time.* Settling time refers to the time required for the system to reach the stable equilibrium. The settling time can be guaranteed to be lower than a specific value when the controller is designed.
- *Robustness.* A robust control system converges to the setpoint despite the underlying model being imprecise. This is very important whenever disturbances have to be rejected and the system has to make decisions with inaccurate measurements.

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.

To enable the analyzability of of these properties in software-intensive adaptive systems, making use formal verification techniques that are typically employed for software systems (e.g., model checking), we formally characterize some of them in the remainder of this section in terms of temporal logic languages, based on their mathematical definition.

5.1 Stability

The concept of **stability** in control theory is a bit different with respect to the concept of stability that is used in self-adaptive software and similar contexts. A control system is stable even if the error $e(t)$ is not converging to zero, but it is bounded:

$$stby \equiv \forall \epsilon > 0 \ \exists \delta(\epsilon) \mid \|y(0) - y^o\| < \delta(\epsilon) \Rightarrow \|y(t) - y^o\| < \epsilon, \forall t > 0 \tag{1}$$

In addition to the bounding of the error $e(t)$ required for stability (captured in the expression above as the norm of the difference between the output and the setpoint $\|y(t) - y^o\|$), **asymptotic stability** is a stronger notion of stability that introduces an additional constraint related to the convergence of the error to zero:

$$astby \equiv stby \wedge \|y(t) - y^o\| \to 0, \text{ for } t \to \infty \tag{2}$$

Figure 5 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 casting into a temporal formula the constraints imposed by the definition stability in the continuous case given in Expression 1. Such characterization can be given on a quantized version of the variables and constants required to define the notion of stability captured in the continuous case:

$$[stby] \equiv \|y_q - y_q^o\| < \delta_q \Rightarrow \Box(\|y_q - y_q^o\| < \epsilon_q) \tag{3}$$

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., $y_q(t) \equiv quant(y(t))$, c.f. Section 2.3). Moreover,

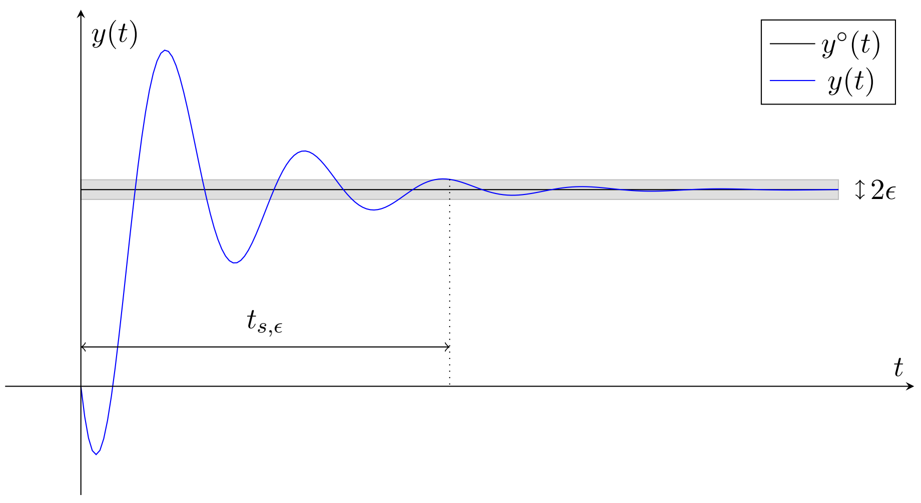


Figure 5: Example of system output stabilization.

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 is referring to the value of $y_q(t)$ in all the states of the discrete timeline (i.e., $y_q(t)$ when $t = 0, t = \tau, t = 2\tau, \ldots$).

For asymptotic stability, we encode the definition of stability we employ in Expression 3, but we add an extra term to the consequent of the implication stating that the system will eventually reach a state from which the error will be bound by the minimum value that we can represent in the quantized version of the variable (i.e., its discretization parameter denoted by η_y):

$$[astby] \equiv \|y_q - y_q^o\| < \delta_q \Rightarrow (\Box(\|y_q - y_q^o\| < \epsilon_q) \wedge \Diamond\Box(\|y_q - y_q^o\| < \eta_y)) \tag{4}$$

This characterization is weaker than the actual notion of stability, and can be considered analogous to a more stringent version of non-asymptotic stability in which the error is bound by the finest granularity that can be distinguished in the discrete model.

5.2 Settling Time

Dynamic performance captures *how* the system is reaching the goal, so it accounts for the transient towards a steady-state. The dynamic performance can be associated with different key indicators, one of which 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:

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

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 (c.f. Expressions 3 and 4), 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

$[\mathsf{time}] \equiv (true, \tau)$ that accrues the time quantum employed for time in the discrete model whenever a transition in the discrete timeline is taken:

$$[t_{s,\epsilon}] \equiv \mathsf{R}^{[\mathsf{time}]}_{=?}[\Diamond\Box\|y_q - y_q^o\| < \epsilon_q] \tag{6}$$

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 in PRCTL assign an infinite reward as the value that is obtained when the expression is quantified. These two aspects make this characterization consistent with the definition given in Expression 5, which characterizes 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.

5.3 Performance

A performance index is a quantitative measure of the performance of a system. It is chosen so that emphasis is given to the important system specifications. A system is considered an optimum control system, when the system parameters are adjusted so that the index reaches an extremum value, commonly a minimum value.

There are several performance indices, and they are always based on the behavior of the error $e(t)$. We consider here the Integral of the Square of the Error (ISE) as a representative performance index to characterize using temporal logic.

$$\text{ISE} \equiv \int_0^T e^2(t)\mathrm{d}t \tag{7}$$

ISE integrates the square of the error over time (see Figure 6). ISE will penalize large errors more than smaller ones (since the square of a large error will be much bigger). Control systems specified to minimize ISE will tend to eliminate large errors quickly, but will tolerate small errors persisting for a long period of time. Often this leads to fast responses, but with considerable, low amplitude, oscillation.

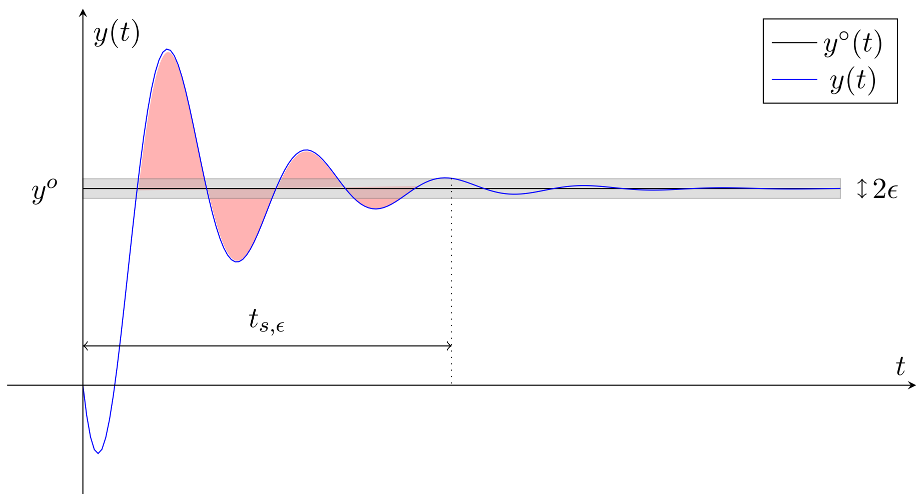


Figure 6: Illustration of integral squared error.

Characterization in Temporal Logic. Similar to settling time, ISE is a quantitative property and therefore we characterize it as a temporal logic expression employing 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 $[\mathsf{error}] \equiv (true, (\|y_q - y_q^o\|)^2)$ that accrues the square of the instantaneous error whenever a transition in the discrete timeline is taken.

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

$$[ISE] \equiv \mathsf{R}^{[\mathsf{error}]}_{=?}[\Diamond\Box\|y_q - y^o_q\| < \epsilon_q] \tag{8}$$

6 Self-Adaptive System Properties

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 characterize performance as an example of non-functional concern in a self-adaptive system, employing an adapted version of the integral squared error (ISE, Section 5.3) as a property to indicate how well the system adapts after a disturbance and achieves stability again.

We assume that RUBiS is working on steady state, but suddenly receives a spike on the request arrival rate that causes the average response time r to go above the threshold T (Figure 7). After violating the threshold, the system adds a server to drive down the response time below T. Before the system stabilizes, the response time may experience some oscillations that make r go above and below T several times. For simplicity, we assume that the setpoint $y^o = T$, although this is not necessarily true in the general case.

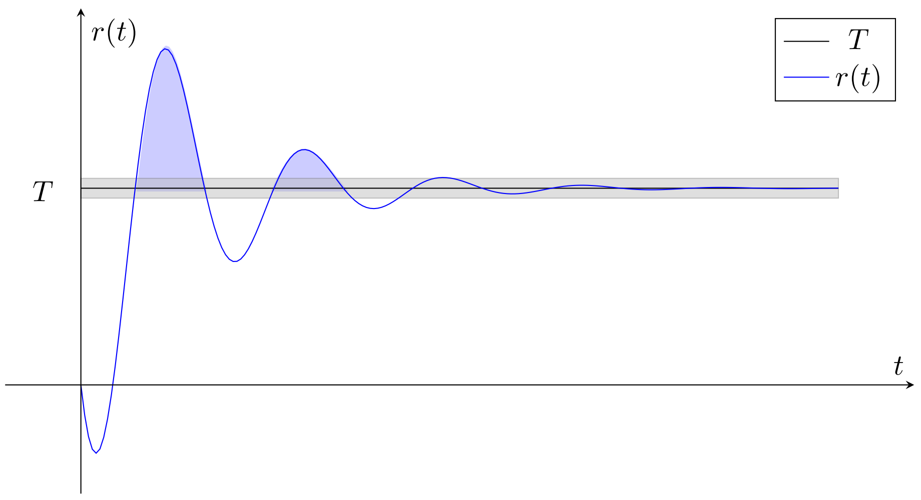


Figure 7: Example of RUBiS performance response with accrued positive squared error.

To obtain an indication of how well the system is adapting, we can employ an adapted version of the integral squared error (ISE) property described in Section 5.3.

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 $r > T$:

$$[\mathsf{penalty}] \equiv (r > T, (r - T)^2) \tag{9}$$

Then, we can employ an expression analogous to Expression 10 to quantify the accrued penalty while the system adapts:

$$\mathsf{R}^{[\mathsf{penalty}]}_{=?}[\Diamond\Box\|r - T\| < \epsilon_q] \;\equiv\; \mathsf{R}^{[\mathsf{penalty}]}_{=?}[\Diamond\, t = [t_{s,\epsilon}]] \tag{10}$$

We can observe that the accrued error corresponds to the colored areas enclosed by T and $r(t)$ in the Figure 7. 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 Section 5.3.

7 Roadmap

In this document, we have identified key properties in control that can constitute a valuable resource to quantify relevant aspects of non-functional run-time behavior in self-adaptive systems. Furthermore, we have discussed the kind of abstractions that can help us bridge the gap between the world of continuous system dynamics in which control properties are typically characterized, and the world of discrete state spaces on which self-adaptive system attributes are measured. Basing on these abstractions, we have presented a possible characterization of a core set of key control properties captured in temporal logic languages that can be employed as input for off-the-shelf run-time verification tools and model checkers. Finally, we discussed an example of how the characterization of control properties that we have given can be adapted to capture properties of concerns in the context of self-adaptive systems.

The material in this document covers an exploratory effort that tackles two of the broad initial goals set to bridge the gap between control and self-adaptive system properties (identification and characterization).

However, our long term goal is understanding if control theory can be used as a formal foundation for self-adaptation, and if so, under what conditions it can be applied. To move towards that goal, further work is needed in terms of identifying correspondences and complementarities between properties in control and self-adaptive systems.

Apart from mapping such correspondences and complementarities, our next steps will involve identifying use cases for properties both on the control and self-adaptation sides, as well as exploring them on an end-to-end application in a real system.

Other related items for further discussion that dovetail with some of the questions discussed in this document include:

- Real-time guarantees in self-adaptive systems. Is it possible to adapt parts of the theories and mechanisms employed in control to provide real-time guarantees in self-adaptive systems operating under different types of uncertainty?
- Formal assessment. Can we employ the characterization of control properties to formally assess the correctness of implementation of controllers? What kind of guarantees can we provide about controllers employing formal verification tools?
- Reconciling multi-step adaptation and control properties. Some control properties like stability only make sense when inputs to the system are fixed. However, self-adaptive systems are typically subject to changing conditions during which the system needs to perform multiple changes on the controlled system that may include driving it to a quiescent state before further changes can be applied, and ultimately, the system can be stabilized. For an effective use of control theory in self-adaptation, it is important to understand how some properties in control map to self-adaptive systems that involve complex multi-step adaptations.
- Transparency. Understanding the rationale for system adaptation. An important emerging quality of autonomous systems (including self-adaptive systems) is the ability to know what the system is doing and why. Rather than functioning as a black box it is critical for such as system to be able to "explain" its actions. Thankfully, when using formal models (either from control theory or from adaptive systems) the system should be able to translate its calculations of control actions into terms that an operator of the system would understand. Finding systematic ways to explain the kinds of models, properties and actions considered in this report require additional research.
- Evolvability/Openness. In the context of software-intensive adaptive systems, change is the rule not the exception. Changes can include requirements for the system, preferences on system qualities, possible control actions that might be taken, and new operating conditions. In many cases such changes cannot be forseen at system design time. Hence it is important for adaptive

systems to be easily evolved. This raises the question of how best to architect adaptive systems to enable ease of change. A critical component of that is keeping a system open to further modification, although the nature of that openness is an important research question.

References

[1] Rice University Bidding System. http://rubis.ow2.org.

[2] S. Andova, H. Hermanns, and J. Katoen. Discrete-time rewards model-checked. In *Formal Modeling and Analysis of Timed Systems: First International Workshop, FORMATS*, volume 2791 of *LNCS*, pages 88–104. Springer, 2003.

[3] K. Åström and R. Murray. *Feedback Systems: An Introduction for Scientists and Engineers.* Princeton University Press, 2010.

[4] R. de Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. R. Schmerl, G. Tamura, N. M. Villegas, T. Vogel, D. Weyns, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. J. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. M. Göschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, A. Lopes, J. Magee, S. Malek, S. Mankovski, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezzè, C. Prehofer, W. Schäfer, R. D. Schlichting, D. B. Smith, J. P. Sousa, L. Tahvildari, K. Wong, and J. Wuttke. Software engineering for self-adaptive systems: A second research roadmap. In R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, editors, *Software Engineering for Self-Adaptive Systems II - International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers*, volume 7475 of *Lecture Notes in Computer Science*, pages 1–32. Springer, 2010.

[5] A. Filieri, M. Maggio, K. Angelopoulos, N. D'Ippolito, I. Gerostathopoulos, A. B. Hempel, H. Hoffmann, P. Jamshidi, E. Kalyvianaki, C. Klein, F. Krikava, S. Misailovic, A. V. Papadopoulos, S. Ray, A. M. Sharifloo, S. Shevtsov, M. Ujma, and T. Vogel. Control strategies for self-adaptive software systems. *TAAS*, 11(4):24:1–24:31, 2017.

[6] H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. *Formal Aspects of Computing*, 6(5):512–535, 1994.

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

[8] M. Z. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In *Formal Methods for Performance Evaluation, 7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM*, volume 4486 of *LNCS*, pages 220–270. Springer, 2007.

[9] A. Simpkins. System identification: Theory for the user, 2nd edition (ljung, l.; 1999) [on the shelf]. *IEEE Robotics Automation Magazine*, 19(2):95–96, 2012.