ISSN 2186-7437

NII Shonan Meeting Report

No. 2017-10

The 2nd Controlled Adaptation of Self-Adaptive Systems Workshop CASaS2017

David Garlan
Nicolas D'Ippolito
Kenji Tei

July 24–28, 2017

National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-Ku, Tokyo, Japan

The 2nd Controlled Adaptation of Self-Adaptive Systems
CASaS2017

Organizers:
David Garlan (Carnegie Mellon University)
Nicolas D'Ippolito (Universidad de Buenos Aires)
Kenji Tei (National Institute of Informatics)

July 24–28, 2017

Self-adaptive systems are required to adapt its behaviour in the face of changes in their environment and goals. Such a requirement is typically achieved by developing a system as a closed-loop system following a Monitor-Analyse-Plan-Act (MAPE) scheme. MAPE loops are a mechanism that allows systems to monitor their state and produce changes aiming to guarantee that the goals are met. In practice it is often the case that to achieve their desired goals, self-adaptive systems must combine a number of MAPE loops with different responsibilities and at different abstraction levels.

Higher-level goals require decision-level mechanisms to produce a plan in terms of the high-level system actions to be performed. Various mechanisms have been proposed and developed for automatically generating decision-level plans (e.g., event-based controller synthesis), providing guarantees about the satisfaction of hard goals (e.g., providing a certain level of service), and supporting improvements in soft goals (e.g., doing this in an efficient or cost-effective manner). These decisions are often made at a time scale of seconds to minutes.

Lower-level goals, on the other hand, typically require control mechanisms that sense the state of the system and environment and react at a fine time granularity of milliseconds. Solutions to this problem are typically based on classical control theory techniques such as discrete-time control.

A successful adaptive system, then, must find ways to integrate these multiple levels of control, leading to an important question of how best to do that, and what concepts. Additionally, concepts from classical control theory (typically applied at low levels of control) can also be useful in understanding higher-level control.

Recently the software engineering community has begun to study the application of control theory and the formal guarantees it provides in the context of software engineering. For example, the 2014 Dagstuhl Seminar "Control Theory meets Software Engineering", is an example of such recent interest. That seminar discussed a variety of possible applications of control theory to software engineering problems.

Also, and perhaps more relevant, is the first CASaS Shonan seminar held in 2016. The seminar focused on formal guarantees that can be provided in self-adaptive systems via the use of control theory (e.g., event-based controller synthesis and discrete-time control). The seminar was a success in many respects. It had over 30 attendees from more than 10 countries. The seminar was an active gathering of outstanding researchers in both control theory and software engineering, and provided a forum in which discussions on the connections between control theory and software engineering for self-adaptive systems could be held. Most of the attendees expressed their intention to continue studying and discussing the relation between control theory and software engineering, which was highlighted as key to address with the requirements of self-adaptive systems.

As in the first edition we expected to involve a group of active researchers in key areas such as Self-Adaptive Systems, Control theory, Game theory, Software Engineering, and Requirements Engineering, creating an ideal environment to discuss current and future applications and possibilities of control theory as a mechanism to provide formal guarantees for self-adaptive systems (e.g., convergence, safety, stability). Encouraged by the success of the first CASaS, we expected to have a number of participants from a wide variety of research areas to further explore the benefits of incorporating the application capabilities and formal framework provided by control theory to self-adaptive systems.

Among the research questions that we expected to discuss are: How to coordinate multiple levels of adaptive control? What kinds of properties from classical control theory can be applied at higher levels to guarantee certain properties? To what extent does the domain and contest of use influence the design of a control regime for adaptation? In what ways can AI techniques of planning and machine learning be applied to adaptive systems? How can one deal with uncertainty in a systematic fashion? How can control theory inform our decisions about ways to incorporate humans into self-adaptive systems?

We envisaged the 5-day meeting to be organised in two main parts. During the first day, participants presented their background and what they are interested in, and there were three lectures about continuous control, discrete-event control, and hybrid approach were given. Then, for the remaining four days, we identified and discussed the most relevant topics selected by the participants in working groups. In the end, we decided to discuss about two topics: "cooperation and coordination" and "properties". The first topic is concerned with ways to incorporate components with ''classical" control implementation into larger systems, which will typically be a mixture of discrete and continuous control, and may need to adapt at an architectural level at run time in response to environmental conditions. The second topic is concerned with ways to formalize properties that are used in control theory in terms that would be useful for systems that reason in terms of discrete control. We divided into two groups, discussed the topics, and created draft reports about the discussion. These reports were further edited and improved, and now constitute the main body of this report.

List of Participants

- Martina Maggio, Lund University, Sweden
- Nir Piterman, Univerity of Leicester, UK
- Thomas Vogel, Humboldt-Universität zu Berlin, Germany
- Alessandro Vittorio Papadopoulos, Mälardalen University, Sweden
- Hiroyuki Nakagawa, Osaka University, Japan
- Javier Camara, Carnegie Mellon University, USA
- Joel Greenyer, Leibniz Universität Hannover, Germany
- Yasuyuki Tahara, The University of Electro-Communications, Japan
- Masako Kishida, National Institute of Informatics, Japan
- Alberto Leva, Politecnico di Milano, Italy
- Shihong Huang, Florida Atlantic University, USA
- Alan Colman, Swinburne University of Technology, Australia
- Amel Bennaceur, The Open University, UK
- Danny Weyns, KULeuven/Linnaeus, Belgium
- Paul Harvey, National Institute of Informatics, Japan
- Lukas Esterle, Aston University, UK
- Patrizio Pelliccione, University of Gothenburg | Chalmers University of Technology, Sweden
- Romina Spalazzese, Malmö University, Sweden

Meeting Schedule

Check-in Day: July 23 (Sun)

- Welcome Reception

Day1: July 24 (Mon)

- Lightning Self-Introduction
- Mini-lecture 1: A Visit to the Control Zoo, Alberto Leva
- Mini-lecture 2: DES and Reactive Synthesis, Nir Piterman
- Mini-lecture 3: Bridging Continuous and Discrete Control, Alessandro Papadopoulos

Day2: July 25 (Tue)

- Topic selection and group building
- Group discussion
- Synchronization

Day3: July 26 (Wed)

- Group discussion
- Synchronization
- Excursion and Main Banquet

Day4: July 27 (Thu)

- Group discussion
- Report writing
- Synchronization

Day5: July 28 (Fri)

- Group discussion
- Report writing
- Wrap up

Group 1

Composition

Composition and Cooperation of Multiple Control Strategies: Automating Control Switch with High-Level Guarantees

Martina Maggio, Nir Piterman, Joel Greenyer, Alberto Leva, Alan Colman, Amel Bennaceur, Paul Harvey, Lukas Esterle, Patrizio Pelliccione, Romina Spalazzese, Nicolas D'Ippolito, David Garlan

1.1 Introduction

By being able to adapt to our environments, humans have been able to not just survive, but to thrive. It is time for software to be able to do the same. As computations find themselves operating in the real world, it is now necessary for them to be able to interpret the world around them, reacting and adapting as necessary. By doing so, such systems become more versatile and useful. The ability for these systems to do this autonomously is required to achieve scalability.

The control of systems and their reactions to their environments is well studied in control theory. Rather than going back to first principles, the use of control theory is perfectly placed to aid and complement the future directions of such self adaptive systems research.

The Controlled Adaptation of Self Adaptive Systems (CASaS) Shonan meeting has provided the time to enable researchers from the Software Engineering (SE) and Control Engineering (CE) communities to determine future research challenges and opportunities in the area of CASaS. Some of these challenges are introduced in this document, with a particular focus on the appropriate organisation of control in adaptive systems.

To more fully explore the fruitfulness of these research challenges, one particular topic – *Guarantees and Assumptions of the Combination of Multiple Controllers* – has been expanded upon. In particular the group focused on developing an approach whereby a model-driven discrete controller could guarantee high-level objectives by switching between a number of continuous controllers that control the plant in various operating regions. By working together, the SE and CE members of the group were able to better understand the issues faced by the respective researchers, both in isolation and with respect to the problem at hand.

The initial part of the meeting discussed the broad challenges involved in integrating research from the SE and CE communities. These topics include abstracting controllers and controlled components so they are readily composable, distributed control, ad hoc control synthesis, handling disruptive and emergent behaviour, and how discrete control techniques from SE can be combined with the continuous control techniques common in CE. The identified problems and challenges are outlined in the last section of this report.

Within this broader set of research challenges, we then focused on one aspect of combining multiple control strategies, namely how discrete control can be used to select and switch between multiple continuous controllers that have been developed to handle both varying operating conditions and varying control objectives.

1.2 Switching Between Controllers at Runtime

Among all the challenges that we have identified, we discussed one specific problem, which is the switch between different controllers at runtime. This constrained our discussion to a general subset of the systems and of the challenges that we may face when multiple systems have to cooperate. In this section, we motivate the need for control switching at runtime.

The need to switch controller during runtime may arise due to changes in operation conditions or due to a change in the goals that the controller is not aware of. For example, an increase in wind speed for a drone or start of rain for a car. Such occurrences may change the operation conditions and force a switch to a controller that provide different guarantees or operates in a different manner. A drone may be scanning an area in fast flight and may be required to switch to a more stable controller in order to investigate something that the drone detected. As another example, a robot may need to change from visual navigation to tracking (a wall, for example) due to a decrease in power.

Moreover, we would also like to avoid continually tuning controllers and readjusting them to match the specific operational conditions. From this standpoint, two different simple controllers that implement the same control logic with two different sets of parameters (for example the gains of a Proportional, Integral and Derivative controller [1]) can be seen as two completely different controllers that the high-level logic may want to switch between. One of these two controllers may be more aggressive in responding to the current error (higher proportional gain), while the other is more conservative (lower gain). Two controllers can have the same code, with different parameters, and for the sake of the analysis, these can be seen as two completely different entities.

Another limitation of simple controllers is that they assume a linear response of the system across its operating region. However, many systems, particularly software systems, exhibit significant non-linear behaviour making them difficult to control using standard continuous-control techniques. For example, a Web Service that is being controlled for response time might automatically change the plant (scale out with extra VMs) in response to demand. One approach to handling this problem is to segment the operating region into a number of smaller domains, each of which can better approximate linear behaviour and have its own controller. This, however, then raises the problem how to implement a higher-level control that switches between these controllers while maintaining desirable stable system-level behaviour.

Instead of having a single controller at hand, our proposed system has a comprehensive set of controllers at hand. While this allows for dealing with various situations and non-linearities, it also gives rise to the question on how to switch between these different controllers at runtime. Switching controllers requires a clear description of the capabilities, assumptions, and guarantees of the individual controller. Analysing this information allows us to identify *areas of operation*, defining the validity of each individual controller given the plant's current state. Furthermore, this enables us to pinpoint those states that enable transition between different controllers, and hence, define a high level discrete state machine for transition. Having such a capability allows the system not only to cope with different, potentially unforeseen, situations but also to reflect on its own performance and therefore determine the *most efficient* controller during runtime.

In order to achieve this, we first require an extensive set of controllers able to handle a wide variety of situations. Each individual controller must be described in a comprehensive fashion allowing for identification of the *operation regions*. In addition, the description has to include assumptions and guarantees such as potential overshoot, settling, and dwell times. Having this information allows us to determine overlapping areas in the regions of operation, i.e. situations or states where multiple controllers will provide valid control output, yielding expected behaviour of the system for a given input. From here, we can define transition strategies, that is represented as state machine describing which controller is used in what situation and what triggers the transition to another controller. Optimally, this is achieved through automatic analysis of the description and the respective regions of operations and overlap. The result is a system operating with an initial controller able to handle simple online variations. Introducing high level switching strategies enables the ability to change controllers in case the situation changes and makes the current controller inappropriate.

1.3 Background

1.3.1 Principles of designing physical controllers

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

The technique that requires the least amount of information is called *synthetic design*. Synthetic design consists in taking pre-designed control blocks and combining them together. It often relies on the experience of the control specialist, who look at experiments performed on the system to be controlled and then decide upon which blocks are necessary. For example, when the output signal is very noisy, a block to be added could be a filter to reduce the noise and captures the original signal. Synthetic design usually starts with a basic control block and adds more to the system as more experiments are performed. Although the information required to set up the control strategy is very low, the expertise necessary to effectively design and tune such systems is high, and both controller design experience and domain-specific knowledge are required. Despite not requiring much information, the formal guarantees that this technique offers are limited [2]. This is due to the empirical nature of the controller's design, where trial and error is applied and elements are added and removed. The main obstacle to formal guarantees is the interaction between the added elements, which is hard to predict a priori.

The technique that requires the most amount of information is often referred to as *analytical design*, and it is based on the solution of an analytical problem [3]. The amount of necessary information greatly increases, since a model of the controlled entity is required. Given on the equation-based model, the controller synthesis selects a suitable equation to link the output variables to the control variables. Depending on which analytical problem is used (the optimisation of some quantities, the tracking of a setpoint, the rejection of "disturbances"), different guarantees are enforced with respect to the controlled system.

In the following, we will use the term "disturbance". A disturbance is something that affect the behaviour of the system under control during the normal operation. For example, suppose we have a drone that is using a control system that is trying to make the drone fly keeping a precise altitude setpoint. The engine's throttle of the motors of the drone determines the height and during the operation the controller is capable of determining a specific value for the throttle that ensures that the drone flies at the prescribed altitude. In this scenario, an example of a disturbance is a gust of wind. The disturbance affects the behaviour of the drone and - ultimately - its altitude. The amount of throttle that must be applied then changes, to reflect the effect of the wind. Depending on the wind direction, the necessary force to be applied to compensate for the wind effect could be different. Controllers can be designed with the aim of rejecting disturbances.

1.3.2 Principles of synthesising software/discrete controllers

Synthesis of discrete event systems is a form of planning that supports decision making in a situated dynamic settings in which programming becomes a difficult or expensive endeavour. The problem of automatically synthesising event-based controllers from environment models and qualitative goal specifications has been widely studied [4, 5, 6]. Given a model of the environment's behaviour (E), a set of system goals (G) and a set of controllable actions (A_C), the controller synthesis problem is to automatically generate a controller (C) that only restricts controllable actions and its parallel execution with the environment ($E \| C$) is guaranteed to satisfy the goals ($E \| C \models G$).

Typical approaches use a combination of automata-based and temporal logics for specifying an environment and system goals.

In this work we use labelled transition Kripke structures to describe the behaviour of the environment and the system. Transitions are labelled with names of actions, some of which the system can monitor or control. States have associated propositions which also may be monitored by the system.

Labelled Transition Kripke Structure. A *labelled transition Kripke structure* (LTKS) is $E = (S, A, P, \Delta, v : S \to 2^P, S_0)$, where S is a finite set of states, $A = A_C \uplus A_M$ is the *communicating alphabet* which we assume is partitioned into controlled and monitored actions, P is a set of propositions, $\Delta \subseteq (S \times A \times S)$ is a transition relation, $v : S \to 2^P$ is a valuation function for states, and $S_0 \subseteq S$ is the set of initial states. A trace of E is $\pi = s_0, \ell_0, s_1, \ell_1, \cdots$, where s_0 is an initial state of E and, for every $i \geq 0$, we have $(s_i, \ell_i, s_{i+1}) \in \Delta$. We denote the set of infinite traces of E by $tr(E)$.

The synthesis problem requires a notion of concurrent execution of the controller with the environment, to model such interactions we use the concept of parallel composition.

Parallel Composition. Let $M = (S_M, A_M, P_M, \Delta_M, v_M, S_{M_0})$ and $E = (S_E, A_E, P_E, \Delta_E, v_E, S_{E_0})$ be LTKSs with $A_M = A_C^M \uplus A_M^M$ and $A_E = A_C^E \cup A_M^E$. *Parallel composition* $\|$ is a symmetric operator such that $E\|M$ is the LTKS $E\|M = (S, A_E \cup A_M, P_M \uplus P_E, \Delta, v, S_0)$, where $S = \{(s_e, s_m) \in S_E \times S_M | v(s_m) \cap P_E = v(s_e) \cap P_M\}$, $S_0 = \{(s_e, s_m) \in S | s_e \in S_{E_0} \wedge s_M \in S_{M_0}\}$, $v((s_e, s_m)) = v_M(s_m) \cup v_E(s_e)$, and Δ is the smallest relation that satisfies the rules below, where $\ell \in A_E \cup A_M$:

$$\frac{E \Rightarrow \ell E'}{E\|M \Rightarrow \ell E'\|M}\ \ell \in A_E \setminus A_M \qquad \frac{M \Rightarrow \ell M'}{E\|M \Rightarrow \ell E\|M'}\ \ell \in A_M \setminus A_E$$

$$\frac{E \Rightarrow \ell E', M \Rightarrow \ell M'}{E\|M \Rightarrow \ell E'\|\ell M'}\ \ell \in A_E \cap A_M$$

We restrict attention to states in S that are reachable from S_0 using transitions in Δ.

Discrete event controllers should guarantee the satisfaction of the desired system goals by only restricting controllable behaviour, formally we ground this intuition with the notion of legality, inspired in that of Interface Automata.

Legal LTKS. Given $E = (S_E, A_E, \Delta_E, s_{E_0})$, $M = (S_M, A_M, \Delta_M, s_{M_0})$ LTKSs, and $A_{E_u} \subseteq A_E$. We say that M is a *Legal LTKS* for E with respect to A_{E_u}, if for all $(s_E, s_M) \in E\|M$ the following holds: $\Delta_{E\|M}((s_E, s_M)) \cap A_{E_u} = \Delta_E(s_E) \cap A_{E_u}$

Intuitively, an LTKS M is *Legal* for and LTS E with respect to an alphabet A_{E_u}, if for all states in the composition $(s_E, s_M) \in S_{E\|M}$ hold that, an action $\ell \in A_{E_u}$ is disabled in (s_E, s_M) if and only if it is also disabled in $s_E \in E$. In other words, M does not restrict E with respect to A_{E_u}.

We formally specify the controller goals using a variation of Linear Temporal Logics [7] called Fluent Linear Temporal Logics [8].

Linear temporal logics (LTL) are widely used to describe behaviour requirements [8, 9, 10, 11]. The motivation for choosing an LTL of fluents is that it provides a uniform framework for specifying state-based temporal properties in event-based models [8]. Fluent Linear Temporal Logic (FLTL) [8] is a linear-time temporal logic for reasoning about fluents. A *fluent* Fl is defined by a pair of sets and a Boolean value: $Fl = \langle I_{Fl}, T_{Fl}, Init_{Fl} \rangle$, where $I_{Fl} \subseteq Act$ is the set of initiating actions, $T_{Fl} \subseteq Act$ is the set of terminating actions and $I_{Fl} \cap T_{Fl} = \emptyset$. A fluent may be initially true or false as indicated by $Init_{Fl}$. Every action $\ell \in Act$ induces a fluent, namely $\dot{\ell} = \langle \ell, Act \setminus \{\ell\}, false \rangle$. Finally, the alphabet of a fluent is the union of its terminating and initiating actions.

Let $\mathcal{F}$ be the set of all possible fluents over *Act*. An FLTL formula is defined inductively using the standard Boolean connectives and temporal operators X (next), U (strong until) as follows:

$$\varphi ::= \mathit{Fl} \mid \neg\varphi \mid \varphi \vee \psi \mid X\varphi \mid \varphi U \psi,$$

where $\mathit{Fl} \in \mathcal{F}$. As usual we introduce $\wedge$, F (eventually), and G (always) as syntactic sugar. Let Π be the set of infinite traces over *Act*. The trace $\pi = \ell_0, \ell_1, \ldots$ satisfies a fluent Fl at position i, denoted $\pi, i \models \mathit{Fl}$, if and only if one of the following conditions holds:

- $\mathit{Init}_{\mathit{Fl}} \wedge (\forall j \in \mathbb{N} \cdot 0 \leq j \leq i \rightarrow \ell_j \notin T_{\mathit{Fl}})$
- $\exists j \in \mathbb{N} \cdot (j \leq i \wedge \ell_j \in I_{\mathit{Fl}}) \wedge (\forall k \in \mathbb{N} \cdot j < k \leq i \ \rightarrow \ell_k \notin T_{\mathit{Fl}})$

Given an infinite trace π, the satisfaction of a formula φ at position i, denoted $\pi, i \models \varphi$, is defined as shown in Figure 1.1. We say that φ holds in π, denoted $\pi \models \varphi$, if $\pi, 0 \models \varphi$. A formula $\varphi \in$ FLTL holds in an LTS E (denoted $E \models \varphi$) if it holds on every infinite trace produced by E.

$$\begin{array}{lcl}
\pi, i \models \mathit{Fl} & \triangleq & \pi, i \models \mathit{Fl} \\
\pi, i \models \neg\varphi & \triangleq & \neg(\pi, i \models \varphi) \\
\pi, i \models \varphi \vee \psi & \triangleq & (\pi, i \models \varphi) \vee (\pi, i \models \psi) \\
\pi, i \models X\varphi & \triangleq & \pi, 1 \models \varphi \\
\pi, i \models \varphi U \psi & \triangleq & \exists j \geq i \cdot \pi, j \models \psi \wedge \forall\ i \leq k < j \cdot \pi, k \models \varphi
\end{array}$$

Figure 1.1: Semantics for the satisfaction operator

1.4 The Framework

In this section we propose a framework that enables switching between an arbitrary number of continuous controllers - as distinct from the composition of discrete controllers. Our goal is to solve a control problem that is composed of both continuous control aspects and discrete control objectives. The plant (*i.e.*, the set of objects that have to be controlled) can operate by selecting one controller at a time from a pool of multiple controllers to deal with the continuous goals. These controllers are designed to modify the behaviour of the plant, while achieving slightly different objectives, *e.g.*, minimisation of consumed energy, maximisation of accuracy, minimisation of time to traverse two points in space.

The motivation for the presence of multiple controllers comes from either changing goals (*e.g.* from maximising travel speed to minimising battery consumption) or changing operating conditions, possibly because of a disruptive event. The assumption that we make in our discussion is that there is some high level goal that the individual control strategies are not aware of and cannot be guaranteed by any single control strategy over the operational space. For example, there might be a controller that maximises travel speed with wet asphalt, and another controller that maximises travel speed with dry asphalt. None of the two controllers alone can optimise the speed of the vehicle in all

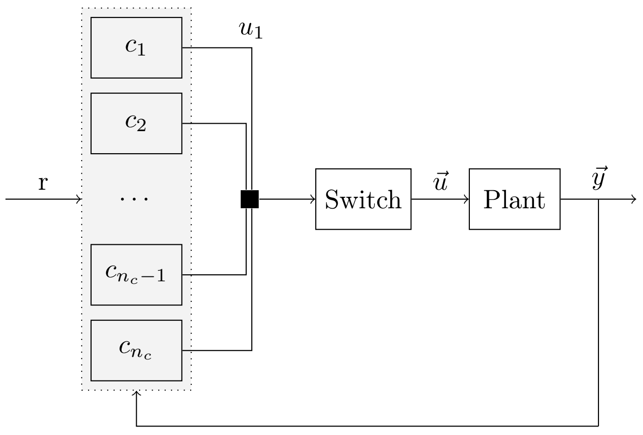


Figure 1.2: The Continuous Control Architecture.

the operating conditions, but the composition of the two controllers can achieve the high level goal of optimising travel speed in all the operational space, with additional knowledge.

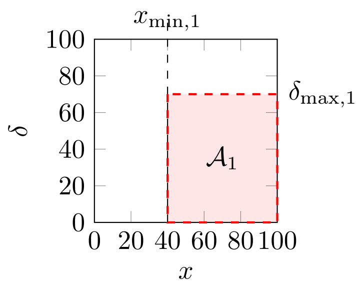
As system designers, we want the plant to expose some guarantees on its behaviour, some of which are control-oriented and some of which are related to the discrete objectives. We denote the set of guarantees that the system has to expose at the global level by $\mathcal{G}_g$.

1.4.1 Continuous Control Design

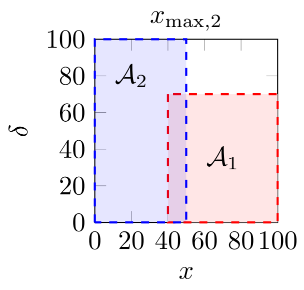
Control Engineers provide a set $\mathcal{C} = \{c_1, c_2, \ldots, c_{n_c}\}$ of n_c controllers, that uses measurements from the plant y, and the reference value (setpoint) r, to produce the control signal u. Figure 1.2 shows the continuous control architecture of the framework, the grey box representing the set of controllers $\mathcal{C}$. Notice that the output of the plant, $\vec{y}$, is a vector and is distributed to the active controller. The active controller may use only some elements of the vector and neglect others, depending on its design. Similarly, the input of the plant – the control signal that the active controller produces, $\vec{u}$ – is also a vector. Some controllers may not prescribe elements of this vector, that are then kept constant during the execution.

A controller c_i is a tuple $c_i = \{\mathcal{X}_i, \mathcal{A}_i, \mathcal{G}_i\}$, where $\mathcal{X}_i$ is the controller code, $\mathcal{A}_i$ is a set of **assumptions** that should be verified for the controller to run properly and $\mathcal{G}_i$ is a set of **guarantees**. Guarantees $\mathcal{G}_i$ are encoded as control properties, *e.g.*, stability, settling time, overshoot [12] and in terms of design concerns, *e.g.*, minimising operational time, minimising energy. The assumptions $\mathcal{A}_i$ of controller i are provided as the region of the operational space in which the guarantees $\mathcal{G}_i$ are valid, *i.e.* the parameters of the system (states and disturbances) that the controller is designed for.

To give a simplified example, assume that the state of the plant to be controlled is the height of a drone, denoted by x and the only disturbance that acts on the plant is the amount of wind, denoted by δ. A controller c_1 may be designed to fly fast at high altitude (optimise speed), but not be resistant to high wind. The operational region of the controller is $\mathcal{A}_1 = \{\delta \leq \delta_{\max,1}, x \geq x_{\min,1}\}$,



(a) $\mathcal{A}_1$ for controller c_1.



(b) $\mathcal{A}_1$ for controller c_1 and $\mathcal{A}_2$ for controller c_2.

Figure 1.3: Illustration of assumption (operational region) for controllers.

where $\delta_{\max,1}$ is a given threshold for the wind and $x_{\min,1}$ is the threshold for the height. Figure 1.3a shows such an assumption region when $\delta_{\max,1} = 70$ and $x_{\min,1} = 40$.

At the same time, the controller c_2 is a slow flying controller, that is resilient to high winds, c_2 is designed to take off and should be used only when the drone's height is less than a prescribed threshold, $\mathcal{A}_2 = \{x \le x_{\max,2}\}$. In our example, $x_{\max,2} = 50$. Figure 1.3b shows the operating regions of both the controllers and displays the overlap between the two.

The operational space $\mathcal{S}$ of the controlled system is defined as the union

$$\mathcal{S} = \cup_{c_i \in \mathcal{C}} \mathcal{A}_i$$

and contains all the possible system operating points for which there exist control solutions. The problem then becomes: given the specification of the set of controllers $\mathcal{C}$, how to synthesise a high-level controller to achieve the objectives in the set of guarantees $\mathcal{G}_g$ that the system is desired to have, both in terms of continuous control guarantees and of high level objectives? Figuratively, this means how to design the logic behind the Switch component in Figure 1.2, that selects the active controller at runtime[1]?

If the control regions $\mathcal{A}_i$ are all disjoint, the problem of designing the high-level controller has a trivial solution. At every point in time, the high level controller should select the single controller c_i for which the current operation point belongs to the region $\mathcal{A}_i$, this being a unique solution. While this simplifies answers to questions like which controller to select in a given situation, it gives rise to the question how to prepare and perform smooth transitions between different controllers. That is, how to prepare different controllers before they are required to actively control the system? Provided that an initialization function is included in the code $\mathcal{X}_i$ of every controller, the transition between different controllers requires calling the initialization function and then the control code at every step. We assume the necessary work to activate a controller can be modelled as a *small* activation delay and, for now, we focus on the assumption that different controllers overlap in their operational space to make the setting

[1]Notice that the logic, here, could be much more complex, including for example the generation of additional controllers and the corresponding assumptions and guarantees at runtime.

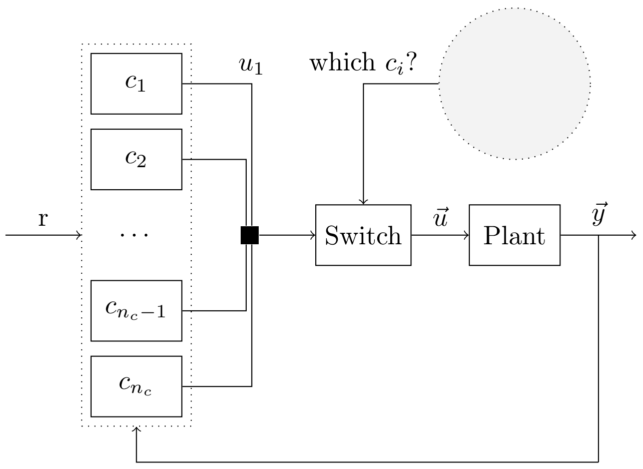


Figure 1.4: The Framework Architecture.

more realistic and the problem interesting. This means further assuming that

$$\exists i, j, i \neq j, \text{s.t.} \mathcal{A}_i \cap \mathcal{A}_j \neq \varnothing$$

i.e., that at least the operational regions of two controllers overlap. Figure 1.4 illustrates the system with the high-level controller that selects between controllers. The grey circle represents the discrete event logic that is the subject of Section 1.4.2 and determines the switching signal.

Finally, for each controller, the system should be kept in the given controller state for a certain amount of time in order to guarantee stability (dwell time). The controller c_i is then complemented with the information ν_i that represent the minimum amount of time to be spent executing it before being able to perform a new switch. The controller then becomes $c_i = \{\mathcal{X}_i, \mathcal{A}_i, \mathcal{G}_i, \nu_i\}$.

1.4.2 Discrete Event Design

The task of the discrete control design is to support the continuous control by supplying the logic for the dotted circle in Figure 1.4. Here we describe how this task would be completed using model-driven development approaches with a framework that could also support automated synthesis from higher-level specifications. We start with an explanation regarding the state space of the discrete controller.

Consider a list of controllers $\mathcal{C} = \{c_1, \ldots, c_{n_c}\}$ as introduced in Section 1.4.1. The controllers $\mathcal{C}$ define an operational space $\mathcal{S} = \bigcup_{c_i \in \mathcal{C}} \mathcal{A}_i$. The different assumptions on the different controllers $\{\mathcal{A}_1, \ldots, \mathcal{A}_{n_c}\}$ induce a partition of $\mathcal{S}$ to *operational regions* based on the controllers that are applicable in a region. That is, for every subset $I \subseteq \mathcal{C}$ the operational region $\bigcap_{c_i \in I} \mathcal{A}_\rangle$ is the region where all the controllers in I are applicable. Clearly, for some subsets $I \subseteq \mathcal{C}$ we have that $\bigcap_{c_i \in I} \mathcal{A}_i = \emptyset$. In such a case, it is impossible to be in a situation where the set of controllers I are all applicable simultaneously.

For effective discrete control, the notion of which controllers are applicable, i.e., those that are currently possible to apply, is a feature of which the higher-level controller needs to be aware. Thus, part of the state of the controller will have to relate to the set I of controllers that are currently applicable. Furthermore, the discrete controller has to "know" which controller is operational at a given time. It follows that the coarsest possible set of states that would enable discrete design would be $\{(I, i) \mid \bigcup_{j\in I} \mathcal{A}_j \neq \emptyset \text{ and } i \in I\}$. That is, the controller should know which controllers are applicable (the set I) and which controller is operational (the controller c_i for $i \in I$). [2]

The description so far does not take the action of the operational controller into account. Indeed, while a controller is operational, it affects and changes some of the measurable parameters relating to the "location" of the plant. It follows that the operational regions above have to be further refined in order to take into account the changes induced due to the operation of the active controller and its dynamics. This refinement needs to take place at runtime and might require the discrete controller to explore the potential region. This gives rise to the question on how to perform such an exploration without jeopardizing the stability of the system?

As an example, consider the case of a drone that has to take off, explore a region (with some low resolution analysis), and when low-resolution analysis discovers something interesting, take high-resolution images. There are available controllers for take-off and landing, for sweeping flight, and for stable flight, which enables high resolution photography. We are ignoring in this example the actual trajectories for sweeping and for the low-resolution analysis. Both flight controllers require a certain height in order to be operational. It follows that when on the ground only the take-off controller is suitable. When applying this controller, the drone increases in height and enters the operational regions for the two flight controllers. From the point of view of the discrete controller, this change is the result of applying the controller but not a change that it applies directly out of its own volition. It may be more appropriate to consider this kind of change as an uncontrollable event that the controller has to be aware of but cannot actively force. On the other hand, once reaching photography altitude, both flight controllers are enabled. It follows that the discrete controller has to take an active decision to switch from the take-off controller to one of the flight controllers. Then, having identified an object that requires further analysis, the discrete controller has to initiate a change of flight controller. In this case, the operational region remained the same, both flight controllers are applicable and the choice which one to actually employ is related to additional information (that is the result of the low-resolution scan). A similar process occurs for landing. There is a high level decision that exploration has ended and an initiation of the landing controller. This happens in an operational region that allows all three controllers (i.e. sweeping flight, stable flight, and landing) to operate. Once the drone lowers down below operational height of the flight controllers there is a perceived (uncontrollable) change of operational region as the flight controllers are no longer applicable.

The refinement of the state space of the discrete controller may depend on the general goal (see below) or on the actual approach to the construction of the discrete controller. For example, when considering the possible changes of state of the discrete-controller, it may be the case that the coarse analysis of the state as done above would not be sufficient. For example, consider an

[2]We store the currently operational controller in the state space of the discrete controller. However, other options are possible, e.g., by considering events that "initialize" controllers and implicitly taking the last controller to have been initialized.

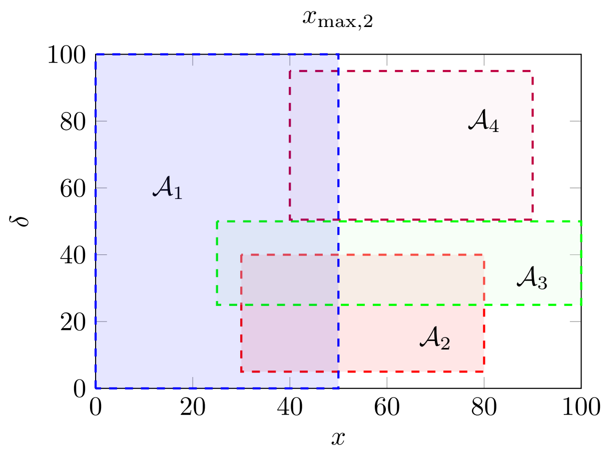


Figure 1.5: Different operational regions for various available controller.

operational region $\mathcal{A}_1$ that borders more than multiple other operational region, namely, $\mathcal{A}_2$, $\mathcal{A}_3$, and $\mathcal{A}_4$. It may be important to distinguish parts of $\mathcal{A}_1$ where operation of controller c may lead to transfer to both neighbours (i.e. $\mathcal{A}_2$ and $\mathcal{A}_3$ as illustrated in Figure 1.5) vs parts of $\mathcal{A}_1$ where operation of c may lead to only one of the neighbours (i.e. $\mathcal{A}_4$). Such distinction would allow to finer choices over which controllers to apply. Indeed, it could be the case that by switching which controller to apply within a region where multiple controllers are possible would enable the discrete controller to drive such a choice. Going back to our drone example, the discrete controller should "know" *a priori* that applying the take-off controller would eventually lead to a situation where either of the two flight controllers is possible to apply.

When considering the features from which the high-level description of the behavior of the plant can be designed we again must consider the operational regions of the controllers. We can treat these operational regions as propositions relating to world state and follow the change of these propositions over time. This goes in both directions. Going top-down, a behavior that is defined by a sequence of truth values of propositions can be extended to possible continuous evolutions of the entire plant. The global correctness of the plant can be deduced from the completion of the discrete trajectories that are possible in the discrete controller with the guarantees over continuous behavior that is provided by the individual controllers applied. Going bottom-up, a continuous behavior can be broken down to the different operational regions that the plant passes through and defines a sequence of propositional values, which can be reasoned about in high level.

1.5 Challenges and future work

Having now discussed the challenge of runtime controller orchestration and high-level design, this section discusses in greater detail the other challenges which were referenced in the introduction. Additionally, this section outlines the possible future directions which may be taken, based on the ideas presented.

1.5.1 Grand challenges integrating SE and CE

Beyond the proposed techniques for switching control, many challenges remain to bridge the gap between (i) the world of Software Engineering,where changeable components/services/behaviors are modularised for reuse and then composed/coordinated, potentially at runtime, to create larger systems, and (ii) the world of Control Engineering where the behaviour of relatively unchanging physical plant can be modelled as black boxes and rigorous analytical control methods applied.

Abstracting Basic Building Blocks. The design of complex systems can be broken down into the design of their basic components. An interesting research direction is the definition of models to abstract the behaviour of control components. This item includes topics such as how to design interfaces between control systems that should belong to a hierarchy, and where the interaction between different components should be designed and engineered. For example, the international standard IEC61499 defines a generic model for a control system function block, which includes data, events, input, and output sources. To properly design the coordination of multiple control components, this specification is lacking some fundamental knowledge, like the assumptions that need guaranteeing for the controller to work properly and the specification of formal guarantees that the controller is capable of providing in case these assumptions are met. Also, the standard does not include runtime reconfiguration. Is it possible to add it? And what if the plant is another piece of software?

Distributed Controllers and Emergent Behaviour. From the design of distributed controllers to the emergence of coordinated behaviour - and - from the desire of a global behaviour to the synthesis of distributed controllers: assuming that a given number of controllers have been synthesised and each of these controllers has a goal and has been verified to fulfil its goal, the interference between different control strategies can still be disruptive. While the benefits of heterogeneous strategies has been shown [13], a remaining research challenges in this case is what can be guaranteed on the behaviour of the "ensemble" of controllers once they are run in a distributed fashion. In a dual manner, it is interesting to understand how to synthesise and/or select and compose at runtime different distributed control loops to achieve a global behaviour, and how to distribute the workload to each of the single controllers. Note that this applies to both the case of configuration and reconfiguration at runtime due to some unforeseen change.

Control of composite systems. Above we make the point that controllers may need to be both componentized and distributed. When a number of controlled SAS systems (as opposed to multiple controllers of a single pre-defined plant) are composed or collaborate, it may also be desirable that the resultant composite entity be also encapsulated as a component. As such its interface would not only express its higher-level function and behaviour but would also expose its aggregated assumptions and control guarantees. For example, a number of heterogeneous drones may form a 'squadron' which can exhibit emergent behaviour beyond the capability of any single drone and for which we want to define high-level goals, guarantees and supervisory interfaces. In such a case, we need to synthesize not only the capabilities of the individual drones but also synthesize their control interfaces. Theories and techniques for such synthesis need to be developed.

Design methodologies. Practical design methodologies need to be developed that integrate SE and CE with their respective formal guarantees. For example, from a practical standpoint, controllers are usually designed following a waterfall approach: requirements are identified, control synthesis is carried out, formal assessment of the system's properties is then verified. If something changes in the specification of the desired behaviour, the design process often should be carried out again. Can software engineering techniques help in reducing the overhead? Also, the design process is usually carried out manually and is error prone (notice that multiple tools and standards are available to support the process). Can the process be automated and/or improved?
Ad-hoc control (existing control strategies learning to cooperate). A common interface (in the form of a system designer, shared knowledge, or of a coordinator) may not always be available. Controllers that "meet" in their working environment should learn how to interact with each other and eventually integrate their behaviours towards a common goal. This might be done in a fully autonomous way, be guided by some indication from the programmer at design time, or be supported by some higher level software entity. This poses many challenges, among which include: (a) the definition of a communication protocol, (b) the definition of common knowledge, (c) the definition of the concept of trust, (d) the definition of the concept of privacy, (e) the concept of non-functional comparability, (f) the ability to deal with the transitory nature of the "meeting". While the controllers have to operate together in an environment, they may want to avoid sharing sensitive information.
Disruptive changes. Usually control systems are designed having in mind a "physically-grounded" use case, which includes boundaries for variables like disturbances. Control theory has found solutions (e.g., robust control), to handle certain degrees of variability at design time and be ready at runtime to react to these variations. The variability may come from different sources (e.g., in the case of a cruise control it may come from a sudden uphill that reduces the car speed or from a motor fault that has a similar effect), provided that their effect has been accounted for in the modelling phase. However, controllers are not able to react to changes that are disruptive to the system and have not been taken into account in the controller design phase. A challenge when dealing with multiple cooperating controllers is to account for disruptive changes and coordinate to handle unforeseen situations. A keyword in control theory is lights off control (control when you can turn off the light - if something disruptive happens, usually the entire plant is turned off by the controller by design). For software, this "lights off control" approach may not be suitable in all approaches, and the challenge is how to achieve this, or be able to move the plant to a situation in which is it achievable. One example of this is a self-driving car. In a situation where the car is faced with an unknown (and detectable) disruption, it will move to the side of the road and stop, as opposed to simply stopping in the middle of the street.

1.5.2 Future Challenges for Discrete Switching Control

The proposed approach to switching control based on identification of operating regions raises a number of further challenges that need to be addressed, particularly with regard to the transition between regions/controllers. It has been assumed that there is an overlap between operating regions to enable the smooth transition between controllers. In the areas where regions overlap there are at least two controllers that are 'good enough' to control the plant. A number of questions follow:

- Can a 'best' controller be determined for such intersects, say, based on respective distance from the boundaries of the plant in the operating space? Is there a general way in which such locations of the plant within the operating space could be modelled and calculated?

- Likewise, can the trajectory of the plant through the operating space be used to predict what controller should be selected, say, based on the velocity vector?

- Can controllers be used to 'drive' the behaviour of the plant away from boundaries with regions that are not controlled, or into regions that better fulfil high-level non-functional requirements of the system?

- Is it necessary that operating regions overlap, or could they be contiguous but disjoint? Can the velocity of the plant through the operating space be used to accurately anticipate the transition point for switching control?

- Is it even necessary that controlled regions be contiguous? If there are white-box models of behaviour in an uncontrolled region it may still be possible to transition between controlled regions via an uncontrolled space. For example, a drone may have controllers for flying in a horizontal orientation either upright or upside-down, but not in a vertical orientation (on edge). Flipping the drone requires that it change from the upright to upside-down controller (or vice-versa), and to be temporarily 'uncontrolled' while on edge. However, the physics of the rotational moment is likely to be able to be well modelled, enabling transition through the uncontrolled space. In this case the controller would deliberately force the plant towards the current region's boundary with an uncontrolled space.

- Operating regions are not just defined by the physical operating conditions but by the control objectives. These objectives may change, necessitating the system be driven to transition between regions. How is such higher-level control to be best achieved?

- Switching controllers will result in a change of behaviour relative to the control objectives. Is there a way to avoid unstable oscillating behaviour? As argued above, one way to enable this is to specify a 'dwell-time' to apply to the time after a new controller has taken effect so as to ensure convergence to the control objective. However, control objectives may be multi-dimensional with no one controller having a globally optimal solution. How can switching of controllers be best controlled in such cases?

- If additional continuous controllers need to be added at runtime, either because of unanticipated environmental conditions or changing requirements, how can the high-level discrete controller be dynamically adapted to incorporate this new operating region?

The robust composition of components is a key concern of SE, with runtime composition (or reconfiguration) being a key concern of SAS. This requires well-defined encapsulated components/services/agents etc.. In the context of the discussion in Section 1.5.1, if we encapsulate, as a self-controlled component, our system consisting of a plant and control-switching mechanisms, what management interface(s) would such a component need? As well as expressing its control characteristics, assumptions, and guarantees, the interface might need to include management operations, for example, to alter set points, change operating modes, or tune continuous control parameters. The interface should also allow the internal discrete controller to interrogate external sensors to determine its 'location' within the operating space. Interfaces for supervisory control or exception handling might also be needed if the component can go, or can anticipate going, into an uncontrolled operating region. If the component is capable of structural adaptation, interfaces would be needed for injecting additional continuous controllers along with appropriate meta-data. More advanced interfaces might enable such self-controlled components to collaborate with other self-controlled components to achieve higher level goals.

1.5.3 Future work

As a first step, the proposed framework for switching control needs to be evaluated. In principle, we would like to devise a case study that has a very small number of tunable parameters, which are easy to relate to the dimensionality/complexity of the case to generate for a particular evaluation test—examples of such parameters can be the number of controllers, each one with its validity region, and the overlapping of the said regions. If this is achieved, not only the satisfaction of high-level goals, but also scalability can be evaluated.

Group 2

Properties

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 Vittorio Papadopoulos, Yasuyuki Tahara, Kenji Tei, Thomas Vogel, and Danny Weyns

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.

2.1 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 [14]. 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 [15].

While obtaining accurate models of non-functional aspects of software behavior can to some extent be achieved using different methods like *system identification* [16], 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 2.2 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 2.3 presents an overview of RUBiS [17], a self-adaptive web multi-tiered system that we employ as a running example to illustrate properties. Section 2.4 presents an overview of key properties in the areas of control and self-adaptive systems, which are later expanded in Sections 2.5 and 2.6, respectively. Finally, Section 2.7 presents a roadmap that discusses directions for future work.

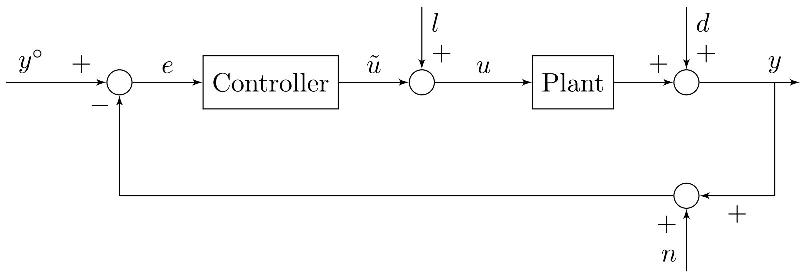


Figure 2.1: Control scheme.

2.2 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.

2.2.1 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 [15].

First, consider the control scheme represented in Figure 2.1.

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 2.1 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 2.1: Names and notation.

2.2.2 Self-Adaptive Systems

We consider the model of a self-adaptive system depicted in Figure 2.2. The **environment** consists of all non-controllable elements that determine the operating conditions of the system (*e.g.*, hardware, network, physical context, etc.).

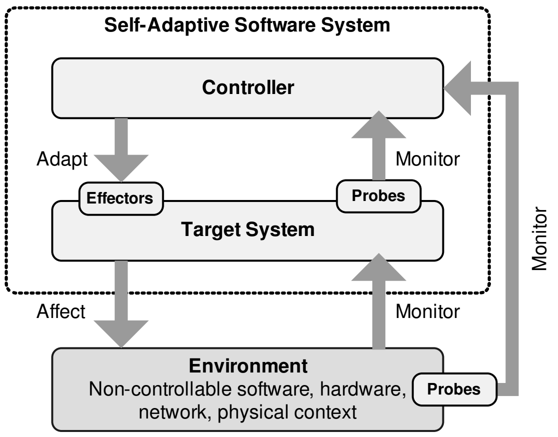

Self-Adaptive Software System
Controller
Adapt
Monitor
Effectors
Probes
Target System
Affect
Monitor
Monitor
Environment
Non-controllable software, hardware, network, physical context
Probes

Figure 2.2: 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 [18].

2.2.3 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:

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

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

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

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 2.3 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]

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 $\iota : [\mathbb{R}^n]_Y \to \mathbb{R}_{\ge 0}$ is a function that assigns rewards to system states, and $\rho : [\mathbb{R}^n]_Y \times [\mathbb{R}^n]_Y \to \mathbb{R}_{\ge 0}$ is a function assigning rewards to transitions.

[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.

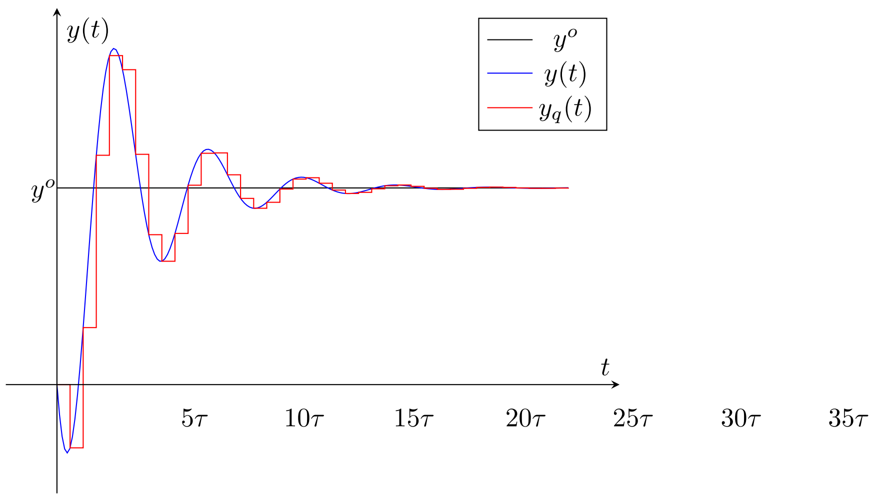


Figure 2.3: Example of discrete quantized vs continuous output.

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.

2.2.4 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 2.2.3).[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 2.2.4), although extensions can be employed to reason explicitly about quantitative aspects of systems that include time or probabilities (Section 2.2.4).

[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.

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.

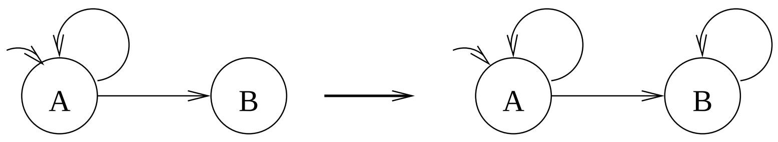
Given a set of atomic propositions AP, any $ap \in AP$ is an LTL formula. Given two LTL formulas ϕ and ψ, then the following are also LTL formulas:

$$\neg\phi \;\mid\; \phi \wedge \psi \;\mid\; \phi \vee \psi \;\mid\; \phi \implies \psi \;\mid\; \phi \Leftrightarrow \psi \;\mid \\ \Box\phi \;\mid\; \Diamond\phi \;\mid\; \bigcirc\phi \;\mid\; \phi \mathsf{U} \psi$$

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 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:

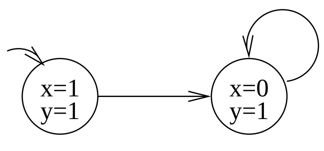
1. A, A, A, . . . (infinitely many A's),
2. A, . . ., A, B, B, . . . (finitely many A's, then infinitely many B's).

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:

[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.



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:

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

Probabilistic Computation Tree Logic with Rewards (PRCTL)

Probabilistic Computation Tree Logic or *PCTL* [19] 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) [20]. 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.

[5]See [21] for details. In the following, we write $Pr_s(\phi)$ as $Pr(\phi)$ for simplicity.

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., $\Diamond\Phi \equiv \mathsf{true}\ \mathsf{U}\ \Phi$, $\Box\Phi \equiv \neg\Diamond\neg\Phi$).

2.3 Case Study

We illustrate our formalization of properties on the Rice University Bidding System (RUBiS) [17], an open-source application that implements the functionality of an auctions website. Figure 2.4 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.

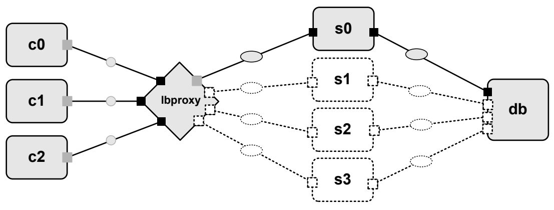


Figure 2.4: RUBiS architecture.

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 [22], 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 2.2). 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.

	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 2.2: Requirements for RUBiS.

2.4 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 2.3) 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 2.3: 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).

2.5 Control Properties

Control systems are usually concerned about four main objectives [14]:

- *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.

2.5.1 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:

$$stby \equiv \forall \epsilon > 0 \; \exists \delta(\epsilon) \mid \|y(0) - y^o\| < \delta(\epsilon) \Rightarrow \|y(t) - y^o\| < \epsilon, \forall t > 0 \qquad (2.1)$$

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:

$$astby \equiv stby \wedge \|y(t) - y^o\| \rightarrow 0, \text{ for } t \rightarrow \infty \qquad (2.2)$$

Figure 2.5 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 2.1. 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:

$$[stby] \equiv \|y_q - y_q^o\| < \delta_q \Rightarrow \Box(\|y_q - y_q^o\| < \epsilon_q) \qquad (2.3)$$

In Expression 2.3, 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 2.2.3). 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.,

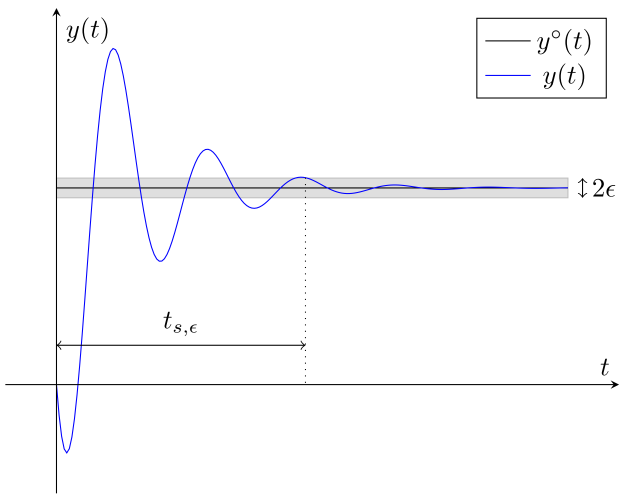


Figure 2.5: Example of system output stabilization.

$y_q \equiv y_q(0)$). 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 $t = 0, t = \tau, t = 2\tau, \dots$).

For asymptotic stability, we encode the definition of stability we employ in Expression 2.3, 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):

$$[astby] \equiv \|y_q - y_q^o\| < \delta_q \Rightarrow (\Box(\|y_q - y_q^o\| < \epsilon_q) \wedge \Diamond\Box(\|y_q - y_q^o\| < \eta_y)) \quad (2.4)$$

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.

2.5.2 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:

$$t_{s,\epsilon} \equiv \inf\{\delta \text{ s.t. } \|y(t) - y^o\| < \epsilon, \forall t \in [\delta, \infty]\} \quad (2.5)$$

In Expression 2.5, 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 2.3 and 2.4), 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:

$$[t_{s,\epsilon}] \equiv \mathsf{R}^{[\mathsf{time}]}_{=?}[\Diamond\Box\|y_q - y^o_q\| < \epsilon_q] \tag{2.6}$$

Expression 2.6 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 2.5, 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.

2.5.3 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.

$$\text{ISE} \equiv \int_0^T e^2(t)\mathrm{d}t \tag{2.7}$$

ISE integrates the square of the error over time (see Figure 2.6). 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.

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 $[\mathsf{error}] \equiv (true, (\|y_q - y^o_q\|)^2)$ that accrues the square of the instantaneous error whenever a transition in the discrete timeline is taken.

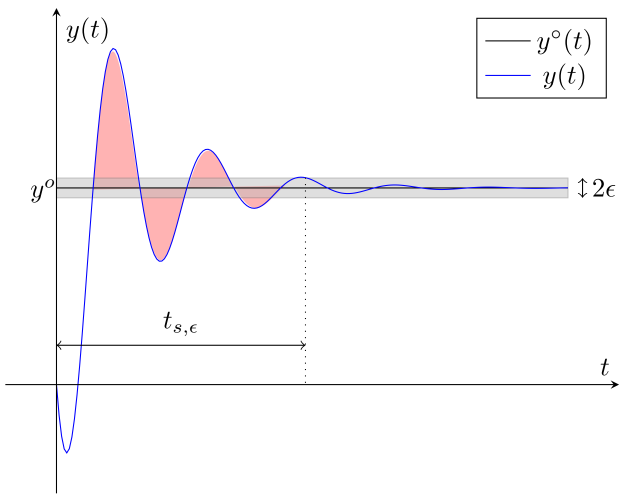


Figure 2.6: Illustration of integral squared error.

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

$$[ISE] \equiv \mathsf{R}^{[\mathsf{error}]}_{=?}[\Diamond\Box\|y_q - y^o_q\| < \epsilon_q] \tag{2.8}$$

2.6 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 2.5.3) 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 2.7). 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.

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 2.5.3.

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 $r > T$:

$$[\mathsf{penalty}] \equiv (r > T, (r - T)^2) \tag{2.9}$$

Then, we can employ an expression analogous to Expression 2.8 to quantify the accrued penalty while the system adapts:

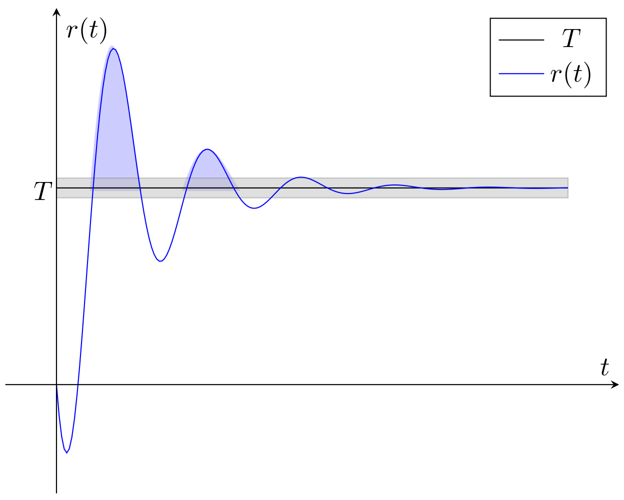


Figure 2.7: Example of RUBiS performance response with accrued positive squared error.

$$\mathsf{R}^{[\mathsf{penalty}]}_{=?}[\Diamond\Box\|r - T\| < \epsilon_q] \;\equiv\; \mathsf{R}^{[\mathsf{penalty}]}_{=?}[\Diamond\, t = [t_{s,\epsilon}]] \tag{2.10}$$

We can observe that the accrued error corresponds to the colored areas enclosed by T and $r(t)$ in the Figure 2.7. 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 2.5.3.

2.7 Roadmap and Future Work

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.

Bibliography

[1] R. Vilanova and A. Visioli. *PID Control in the Third Millennium: Lessons Learned and New Approaches.* Advances in Industrial Control. Springer London, 2012.

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

[3] K.J Åström and R.M. Murray. *Feedback Systems: An Introduction for Scientists and Engineers.* Princeton University Press, 2008.

[4] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. Synthesis of reactive(1) designs. *J. Comput. Syst. Sci.*, 78(3):911–938, 2012.

[5] Nicolás D'Ippolito, Victor Braberman, Nir Piterman, and Sebastián Uchitel. Synthesising non-anomalous event-based controllers for liveness goals. *ACM Tran. Softw. Eng. Methodol.*, 22, 2013.

[6] Amel Bennaceur and Valérie Issarny. Automated synthesis of mediators to support component interoperability. *IEEE Trans. Software Eng.*, 41(3):221–240, 2015.

[7] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '89, pages 179–190, New York, NY, USA, 1989. ACM.

[8] Dimitra Giannakopoulou and Jeff Magee. Fluent model checking for event-based systems. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, pages 257–266, New York, NY, USA, 2003. ACM.

[9] Axel van Lamsweerde and Emmanuel Letier. Handling obstacles in goal-oriented requirements engineering. *IEEE Transactions on Software Engineering*, 26:978–1005, October 2000.

[10] Emmanuel Letier and Axel van Lamsweerde. Agent-based tactics for goal-oriented requirements elaboration. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 83–93, New York, NY, USA, 2002. ACM.

[11] Raman Kazhamiakin, Marco Pistore, and Marco Roveri. Formal verification of requirements using spin: A case study on web services. In *Proceedings of the Software Engineering and Formal Methods, Second International Conference*, SEFM '04, pages 406–415, Washington, DC, USA, 2004. IEEE Computer Society.

[12] Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, Nicolás D'ippolito, Ilias Gerostathopoulos, Andreas Berndt Hempel, Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, et al. Control strategies for self-adaptive software systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 11(4):24, 2017.

[13] Peter R. Lewis, Lukas Esterle, Arjun Chandra, Bernhard Rinner, Jim Torresen, and Xin Yao. Static, dynamic, and adaptive heterogeneity in distributed smart camera networks. *ACM Trans. Auton. Adapt. Syst.*, 10(2):8:1–8:30, June 2015.

[14] Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, Nicolás D'Ippolito, Ilias Gerostathopoulos, Andreas Berndt Hempel, Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, Filip Krikava, Sasa Misailovic, Alessandro Vittorio Papadopoulos, Suprio Ray, Amir Molzam Sharifloo, Stepan Shevtsov, Mateusz Ujma, and Thomas Vogel. Control strategies for self-adaptive software systems. *TAAS*, 11(4):24:1–24:31, 2017.

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

[16] 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.

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

[18] Rogério de Lemos, Holger Giese, Hausi A. Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley R. Schmerl, Gabriel Tamura, Norha M. Villegas, Thomas Vogel, Danny Weyns, Luciano Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Ronald J. Desmarais, Schahram Dustdar, Gregor Engels, Kurt Geihs, Karl M. Göschka, Alessandra Gorla, Vincenzo Grassi, Paola Inverardi, Gabor Karsai, Jeff Kramer, Antónia Lopes, Jeff Magee, Sam Malek, Serge Mankovski, Raffaela Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezzè, Christian Prehofer, Wilhelm Schäfer, Richard D. Schlichting, Dennis B. Smith, João Pedro Sousa, Ladan Tahvildari, Kenny Wong, and Jochen Wuttke. Software engineering for self-adaptive systems: A second research roadmap. In Rogério de Lemos, Holger Giese, Hausi A. Müller, and Mary 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.

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

[20] Suzana Andova, Holger Hermanns, and Joost-Pieter 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.

[21] Marta Z. Kwiatkowska, Gethin Norman, and David 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.

[22] Cristian Klein, Martina Maggio, Karl-Erik Årzén, and Francisco 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.