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

Shonan Seminar #110, Group 1

September 21, 2017

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.

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.

3 Background

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.

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 $\ell \in A_E \cup A_M$:

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

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

Figure 1: Semantics for the satisfaction operator

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 [?] called Fluent Linear Temporal Logics [?].

Linear temporal logics (LTL) are widely used to describe behaviour requirements [?, ?, ?, ?]. 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 [?]. Fluent Linear Temporal Logic (FLTL) [?] is a linear-time temporal logic for reasoning about fluents. A *fluent Fl* is defined by a pair of sets and a Boolean value: $\mathit{Fl} = \langle I_{\mathit{Fl}}, T_{\mathit{Fl}}, \mathit{Init}_{\mathit{Fl}} \rangle$, where $I_{\mathit{Fl}} \subseteq \mathit{Act}$ is the set of initiating actions, $T_{\mathit{Fl}} \subseteq \mathit{Act}$ is the set of terminating actions and $I_{\mathit{Fl}} \cap T_{\mathit{Fl}} = \emptyset$. A fluent may be initially true or false as indicated by $\mathit{Init}_{\mathit{Fl}}$. Every action $\ell \in \mathit{Act}$ induces a fluent, namely $\dot{\ell} = \langle \ell, \mathit{Act} \backslash \{\ell\}, \mathit{false} \rangle$. 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. 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.

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 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 $\mathcal{G}_g$.

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 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 [7] 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 $\mathcal{A}_1 = \{\delta \leq \delta_{\max,1}, x \geq x_{\min,1}\}$, where $\delta_{\max,1}$ is a given threshold for the wind and $x_{\min,1}$ is the threshold for the height. Figure 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, $\mathcal{A}_2 = \{x \leq x_{\max,2}\}$. In our example, $x_{\max,2} = 50$. Figure 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

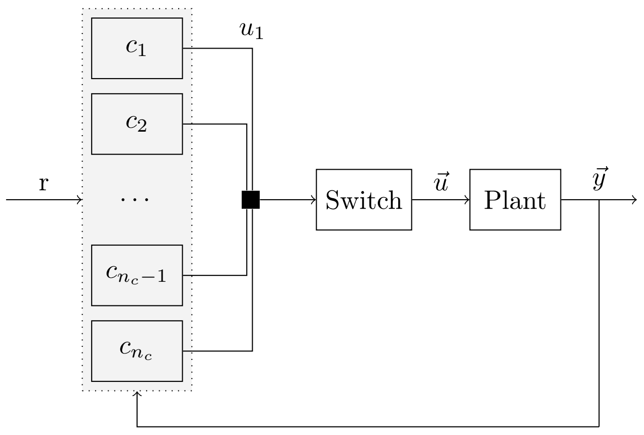
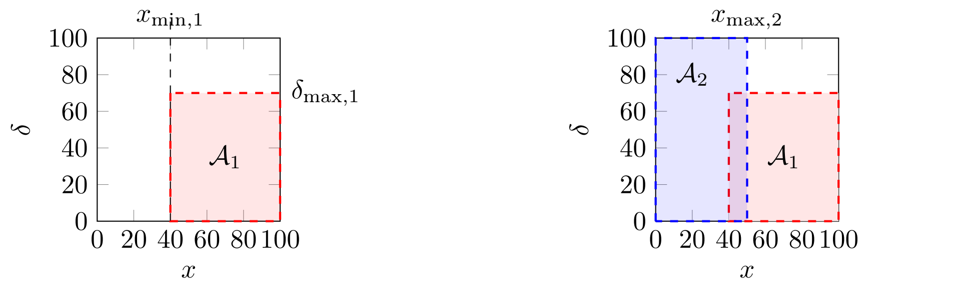


Figure 2: The Continuous Control Architecture.



(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 3: Illustration of assumption (operational region) for controllers.

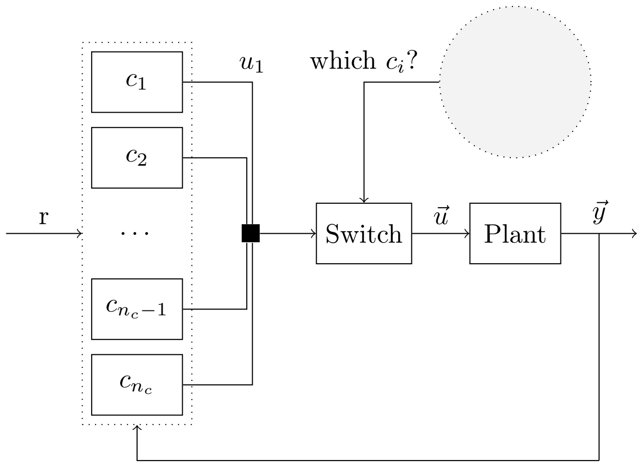


Figure 4: The Framework Architecture.

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 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 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 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 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]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.

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 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 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 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

[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.

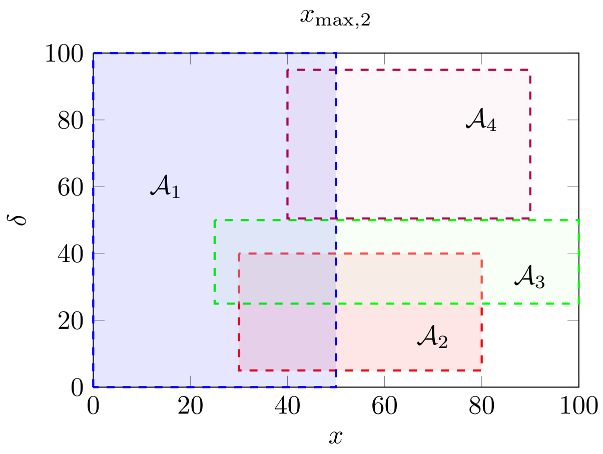


Figure 5: Different operational regions for various available controller.

to transfer to both neighbours (i.e. $\mathcal{A}_2$ and $\mathcal{A}_3$ as illustrated in Figure 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.

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.

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 [8], 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.

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 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.

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.

References

[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] 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.

[8] 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.