

# Provable Correct and Adaptive Simplex Architecture for Bounded-Liveness Properties

Benedikt Maderbacher<sup>1</sup>, Stefan Schupp<sup>2</sup>, Ezio Bartocci<sup>2</sup>, Roderick Bloem<sup>1</sup>,  
Dejan Nickovic<sup>3</sup>, and Bettina Könighofer<sup>1</sup>

<sup>1</sup> Graz University of Technology

<sup>2</sup> TU Wien

<sup>3</sup> AIT Austrian Institute of Technology

**Abstract.** We propose an approach to synthesize simplex architectures that are *provably correct* for a rich class of temporal specifications, and are *high-performant* by optimizing for the time the advanced controller is active. We achieve provable correctness by performing a static verification of the baseline controller. The result of this verification is a set of states which is proven to be safe, called the *recoverable region*. During runtime, our Simplex architecture *adapts* towards a running advanced controller by exploiting proof-on-demand techniques. Verification of hybrid systems is often overly conservative, resulting in over-conservative recoverable regions that cause unnecessary switches to the baseline controller. To avoid these switches, we invoke targeted reachability queries to extend the recoverable region at runtime.

Our offline and online verification relies upon reachability analysis, since it allows observation-based extension of the known recoverable region. However, detecting fix-points for bounded liveness properties is a challenging task for most hybrid system reachability analysis tools. We present several optimizations for efficient fix-point computations that we implemented in the state-of-the-art tool HyPro that allowed us to automatically synthesize verified and performant Simplex architectures for advanced case studies, like safe autonomous driving on a race track.

## 1 Introduction

With the unprecedented amount of available computational power and the proliferation of artificial intelligence, modern control applications are becoming increasingly autonomous. The increasing adoption of the DevOps paradigm by the cyber-physical systems (CPS) community facilitates development of the control systems beyond their deployment – the observation and collection of data during system operation allows control systems to evolve and improve over time.

Developing trusted advanced controllers has therefore become a major challenge in safety-critical domains. Since the high complexity of advanced controllers renders formal verification infeasible, runtime assurance methods [1], which ensure safety by monitoring and altering the execution of the controller, gain more and more importance.

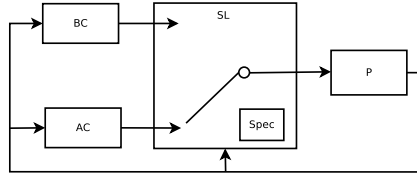


Fig. 1: Schematic of a Simplex architecture

**Simplex Architecture.** The Simplex Architecture [2,3] is a well-established runtime assurance architecture, originally proposed for reliable upgrades in a running control system. Let the *plant*  $P$  be the physical system of hybrid nature (the system’s components exhibit mixed discrete-continuous behavior), and  $\varphi$  be the safety specification. The Simplex architecture that ensures that  $P$  works within the specification  $\varphi$ , consists of three components, as illustrated in Fig. 1:

1. The *baseline controller* (BC) is a formally verified controller with respect to the given specification  $\varphi$ . Therefore, it is proven that the baseline controller provides control inputs to the plant in such a way that  $\varphi$  is satisfied. This holds under the assumption that the plant is initially in a state from which the baseline controller is able to satisfy  $\varphi$ . We call the set of plant’s states, from which the baseline controller guarantees safe operation according to  $\varphi$ , the *recoverable region*.
2. The *advanced controller* (AC), a highly efficient and complex controller that might incorporate deep neural networks. However, due to its complexity, the advanced controller cannot be formally verified and therefore we have no guarantees whether it always satisfies  $\varphi$ .
3. The *switching logic* (SL) monitors the execution of the advanced controller and hands the control to the baseline controller if the plant would otherwise leave the recoverable region where the baseline controller guarantees  $\varphi$ .

The Simplex architecture *guarantees safety* by ensuring that the plant operates within the formally proven, recoverable region of the baseline controller, and facilitates *high performance* by enabling the advanced controller maximal freedom and only restricting its operation to the recoverable region.

**Challenges.** While conceptually simple, the implementation of a Simplex architecture is challenging. To guarantee safety, it is required to formally verify the baseline controller. Hybrid automata [4] are the official model for accurately describing composite systems that combine discrete computational and continuous processes, therefore hybrid automata are the desired model for baseline controllers. However, the verification of hybrid automata is typically the bottleneck due to the inherent limits of exhaustive verification tools. To obviate formal verification, it is common practice to make assumptions about the correctness of a given baseline controller and its recoverable region. By doing so, one loses all correctness guarantees the Simplex architecture should provide.

The second challenge is to implement a Simplex architecture that allows high performance while being formally verified. In case the formal verification of the

baseline controller was successful, the accumulation of over-approximation errors during the static verification phase often results in an over-conservative recoverable region. Therefore, the switching logic gives control to the baseline controller much more often than necessary, causing unnecessary drops in performance.

**Problem Statement.** We start from a given plant and a baseline controller, both modeled as hybrid automata, and a safety specification expressed in signal temporal logic (STL) [5], which includes bounded-liveness requirements. The problem is to synthesize Simplex architectures that are provably correct for the given specification for unbounded time, and are high-performant by optimizing for the time the advanced controller is active.

**Our Approach.** We compute at design time an initial recoverable region as the set of states reachable by the baseline controller from an initial region. During the reachability analysis, we check that the specification is satisfied for all reachable states from the baseline controller starting from its initial states. Using a fixpoint detection on the set of reachable states allows us to guarantee safety with respect to the given specification on infinite time.

We then optimize, during the execution, the performance of the operating Simplex architecture incrementally by adapting towards the running advanced controller. Whenever the advanced controller proposes an output that would cause the plant to leave the current recoverable region, we perform two steps: we switch to the baseline controller, and we try to enlarge the recoverable region. To enlarge the recoverable region, we collect *suspicious states* that would have been reached in case the last command of the advanced controller would have been executed. Next, we analyze the behavior of the baseline controller on all suspicious states. If we are able to prove that the baseline controller ensures safety for a suspicious state, we add this state and all states reachable from it to the recoverable region. Next time the advanced controller tries to enter such a state, the switching logic will not interfere.

**Contributions.** Our main results can be summarized as follows:

- We propose a workflow to synthesize *provable correct* Simplex architectures that *dynamically adapt* to a running advanced controller.
- Our proposed methodology enables a lightweight reachability analysis to show safety of suspicious states on demand. To this end, we employ flowpipe construction based reachability analysis as an inductive approach which allows to compute sets of reachable states for given initial configurations. This enables local updates of the known recoverable region in a light-weight fashion based on current observations of the system in an iterative way enabling us to involve information obtained at runtime. We aim to incorporate already established knowledge about recoverable regions into the verification approach to further improve its scalability.
- To the best of our knowledge, we are the first that synthesize Simplex architectures for bounded-liveness specifications.
- We present several optimizations for efficient fixpoint computations. In particular, we introduce novel data structures for faster lookup as well as set-theoretic and symbolic approaches to improve fixpoint detection. We implemented these optimizations in the hybrid system reachability analysis tool HyPro [6].

Only with our optimizations, we were able to verify hybrid automata against bounded liveness properties in HyPro.

- We present a detailed case study on safe driving on a racing track and the effects of optimizing the Simplex architecture during runtime.

**Related work.** Original works on the Simplex architecture [1–3], as well as many recent work [7–9] assume to have a verified baseline controller and a correct switching logic given. Under these assumptions, the papers guarantee safe operation of the advanced controller. However, these assumptions are very strong and the works ignore the challenges and implied limitations that need to be addressed in order to get a verified baseline controller and a switching logic that is guaranteed to switch at the correct moment. The reason for many works to leave out these steps is that a general safety statement for *unbounded time* for the baseline controller is required (i.e., a fixpoint in the analysis). Depending on the utilized method, fixpoints in the analysis cannot always be found, as some verification methods tend to have bad convergence due to accumulating errors.

Recent works that verify the baseline controller deploy standard methods to verify hybrid systems such as barrier certificates [10–12], and using forward or backward reachability analysis [13]. Our method is independent of the concrete approach that is used for offline verification. In general, for methods based on flowpipe construction for forward or backward reachability analysis, there exists a trade-off between accuracy and complexity: using simple shapes to over-approximate the reachability tubes results in overly-conservative recoverable regions, while using too complex shapes requires difficult computations to check for a fixpoint. By using the concept of proof on demand, we allow simple shapes for the reachability analysis but amend the problem of an over-conservative recoverable region by enlarging the region on demand.

While several works study provable correct Simplex architectures, there is only little work on how to create high-performant Simplex architectures. Similar to our approach, the work in [14] uses online computations to increase performance. While our approach adapts to a given advanced controller and therefore the number of online proofs reduces during exploitation, the approach in [14] performs the same online computations repeatedly.

Furthermore, to the best of our knowledge, no work considered temporal logic properties beyond safety invariants to be enforced by a Simplex architecture. Instead, our work allows us to specify bounded reachability and bounded liveness properties and conjunctions of these properties.

Runtime assurance covers a wide range of techniques and has several application areas, for example enforcing safety in robotics [15] or in machine learning [16]. Runtime enforcers, often called shields, directly alter the output of the controller during runtime to enforce safety. In the discrete setting, such enforcers can be automatically computed from a model of the plant’s dynamics and the specification using techniques from model checking and game theory [17]. In the continuous domain, inductive safety invariants such as a control Lyapunov functions [18] or control barrier functions [19] are used to synthesize runtime enforcers.

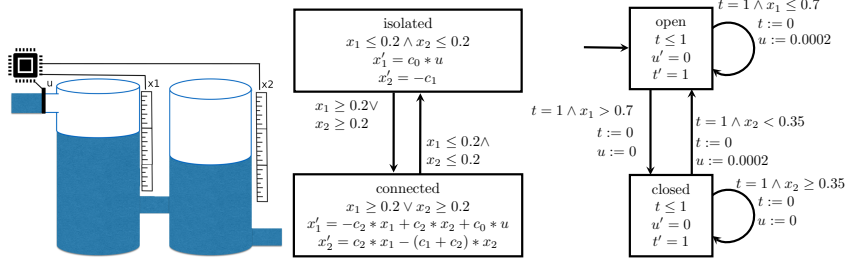


Fig. 2: Left: Illustration of the plant. Middle: Hybrid automaton  $\mathcal{H}_P$  modeling the plant. Right: Hybrid automaton  $\mathcal{H}_{BC}$  implementing the baseline controller.

## 2 Illustrative Example

As an illustrative example, we use a well-known textbook example of a coupled water tank system [20], as illustrated in Figure 2 (left). Using this example, we will outline how we construct the Simplex architecture from a given safety specification in STL and a given baseline controller in form of a hybrid automaton.

**Plant P.** The plant consists of two tanks that are connected by a pipe that is located at a height of 0.2 m from the floor. The left tank has an inflow that can be adjusted by a controller. The right tank has an outflow pipe that constantly drains water. The plant with the two connected water tanks can be modeled by the hybrid automaton  $\mathcal{H}_P$  given in Figure 2 (middle). The automaton has two state variables  $x_1$  and  $x_2$ , corresponding the level of water in the left and right tank respectively, and one control dimension  $u \in [0, 0.0005]$  influencing how much water is added to the left tank. The two states reflect the two modes with different dynamics. If the water in any of the tanks is higher than 0.2 m water can flow through the connecting pipe and the levels in the two tanks equalize. Otherwise, the tanks are isolated and evolve only according to their own dynamics.

**Safety Specification  $\Phi$ .** The safety specification  $\Phi = \Phi_1 \wedge \Phi_2$  requires that the following two properties  $\Phi_1$  and  $\Phi_2$  are satisfied:

1. The water tanks may not be filled beyond their maximum filling height of 0.8 m. This property can be expressed in STL via  $\Phi_1 = \mathbf{G}(x_1 \leq 0.8 \wedge x_2 \leq 0.8)$ .
2. If the water level of the right tanks falls below 0.12 m it has to be filled up to at least 0.3 m within the next 30 time units. This is written in STL as  $\Phi_2 = \mathbf{G}(x_2 \leq 0.12 \rightarrow \mathbf{F}_{[0,30]}(x_2 \geq 0.3))$ .

A safety specification  $\Phi$  in STL can be transformed into a hybrid automaton  $\mathcal{H}_\Phi$ . We discuss this transformation in the next section.

**Baseline Controller.** We use the baseline controller that is given by the hybrid automaton  $\mathcal{H}_{BC}$  in Figure 2.  $\mathcal{H}_{BC}$  consists of two locations *open* and *closed*. At the end of a cycle ( $c = 1$ ) the controller observes  $x_1$  and  $x_2$  to determine whether to stay in its current location or switch to the other one.  $\mathcal{H}_{BC}$  remains in *open* as long as  $x_1 \leq 0.7$  and in *closed* as long as  $x_2 \geq 0.35$ . After every transition, the value of  $u$  is set to either 0 or 0.0002 depending on the target location.

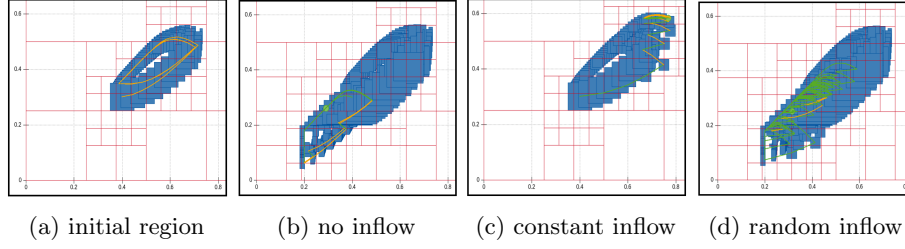


Fig. 3: Visualisation of the initial recoverable region and adapted recoverable regions of the different advanced controllers.

**Static verification for a conservative Simplex architecture.** In the first step, we verify the baseline controller  $\mathcal{H}_{BC}$  acting within the plant  $\mathcal{H}_P$  with respect to the specification  $\mathcal{H}_\Phi$  for the initial states of the plant  $x_1 \in [0.35, 0.45]$  and  $x_2 \in [0.25, 0.35]$ . To compute the initial recoverable region, we compute the fixpoint of reachable states while checking that no bad state defined by  $\mathcal{H}_\Phi$  is contained in the reachable states. We verified the baseline controller using the reachability analysis tool HyPRO [6]. Figure 3(a) illustrates the initial recoverable region. The blue squares depict the initially known recoverable region for each control cycle. The orange trajectory shows the behavior of the plant when controlled by the baseline controller.

**Adapting to the advanced controller via proof on demand.** In the second step, we run the Simplex architecture for 500 control cycles and evolve the recoverable region during runtime. Figure 3(b)-(d) shows the growth of the recoverable region over time for different advanced controllers: the first advanced controller (Figure 3(b)) sets  $u$  to 0 constantly, the second advanced controller (Figure 3(c)) sets  $u$  to 0.0004 constantly, and the third advanced controller (Figure 3(d)) picks  $u \in [0.0001, 0.0005]$  equally distributed with a 10 % probability, and sets  $u = 0$  otherwise. In each figure, the green trajectory shows the behavior of the plant when being controlled by the corresponding advanced controller. The plots only show the known recoverable region at the end of each control cycle. The points where the trajectory exits the blue region these points are between control cycles and have been verified safe, but they are not stored for efficiency reasons. In these experiments, we check the safety of states outside of the recoverable region on the fly and immediately add them to the recoverable region if they are safe. The switching logic only switches to the baseline controller, if the advanced controller would visit unsafe states that cannot be added to the recoverable region. Note, that these proofs on demand can also be performed in the background.

### 3 Background

#### 3.1 Reachability Analysis of Hybrid Systems

We use *hybrid automata (HA)* as a formal model for hybrid systems.

**Definition 1 (Hybrid automata [4]).** A hybrid automaton  $\mathcal{H} = (Loc, Lab, Edge, Var, Init, Inv, Flow, Jump)$  consists of a finite set of locations  $Loc = \{\ell_1, \dots, \ell_n\}$  which represents the discrete states, a finite set of synchronization labels  $Lab$ , which coordinate state changes between several automata, a finite set of jumps  $Edge \subseteq Loc \times Lab \times Loc$ , that determines possible discrete state changes, a finite set of variables  $Var = \{x_1, \dots, x_d\}$ ; a state  $\sigma = (\ell, \nu)$  of  $\mathcal{H}$  consists of a location  $\ell$  and a valuation  $\nu \in \mathbb{R}^d$  for each variable,  $S = Loc \times \mathbb{R}^d$  denotes the set of all states, a set of states  $Inv$  called invariant, which restricts the values  $\nu$  for each location, a set of initial states  $Init \subseteq Inv$ , a flow relation  $Flow$  where  $Flow(\ell) \subseteq \mathbb{R}^{Var} \times \mathbb{R}^d$  determines for each state  $(\ell, \nu)$  the set of possible derivatives  $Var$ , a jump relation  $Jump$  where  $Jump(e) \subseteq \mathbb{R}^d \times \mathbb{R}^{d'}$  defines for each jump  $e \in Edge$  the set of possible successors  $\nu'$  of  $\nu$ .

Every behavior of  $\mathcal{H}$  must start in one of the initial states  $Init \subseteq Inv$ . Jump relations are typically described by a guard set  $G \subseteq \mathbb{R}^d$  and an assignment (or reset)  $\nu' = r(\nu)$  as  $Jump(e) = \{(\nu, \nu') \mid \nu \in G \wedge \nu' = r(\nu)\}$ . For simplicity, we restrict ourselves to the class of *linear hybrid automata*, i.e., the dynamics ( $Flow$ ) are described by systems of linear ordinary differential equations and guards ( $G$ ), invariant conditions ( $Inv$ ), and sets of initial variable valuations are described by linear constraints. Resets on discrete jumps are given as affine transformations. In this work, we use composition of hybrid automata as defined in [21] with label-synchronization on discrete jumps and shared variables.

A path  $\pi = \sigma_1 \rightarrow_\tau \sigma_2 \rightarrow_e \dots$  in  $\mathcal{H}$  is an ordered sequence of states  $\sigma_i$  connected by *time transitions*  $\rightarrow_\tau$  of length  $\tau$  and *discrete jumps*  $\rightarrow_e, e \in Edge$ . Time transitions follow the flow relation while discrete jumps follow the edge and jump relations, we refer to [21] for a formal definition of the semantics. Paths naturally extend to *sets of paths* which collect paths with the same sequence of locations but different variable valuations.

**The reachability problem in hybrid automata.** A state  $\sigma_i = (\ell, \nu)$  of  $\mathcal{H}$  is called *reachable*, if there is a path  $\pi$  leading to it with  $\sigma_1 \in Init$ . The reachability problem for hybrid automata tries to answer whether a given set of states  $S_{bad} \subseteq S$  is reachable. Since the reachability problem is in general undecidable [22], current approaches often compute over-approximations of the sets of reachable states for *bounded reachability*. Note that reachability analysis follows all execution branches, i.e., does not resolve any non-determinism induced by discrete jumps in the model. That means, that computing alternatingly time- and jump-successor states may yield a tree-shaped structure (nodes contain time-successors, the parent-child relation reflects discrete jumps, see also [23]) which covers all possible executions.

**Flowpipe construction for reachability analysis.** For a given hybrid automaton  $\mathcal{H}$ , flowpipe construction (see e.g., [24]) computes a set of convex sets

$$R = reach_{\leq \alpha}^{\mathcal{H}}(\sigma),$$

which are guaranteed to cover all trajectories of bounded length  $\alpha$  that are reachable from a set of states  $\sigma$ . We use  $reach_{=\alpha}^{\mathcal{H}}(\sigma)$  to denote the set of states

that are reached after exactly  $\alpha$  time, and  $reach_\infty^{\mathcal{H}}(\sigma)$  to denote the set of states reachable for unbounded time.

The method over-approximates time-successor states by a sequence of sets (segments), referred to as *flowpipe*. Segments that satisfy a guard condition of an outgoing jump of the current location allow taking said jump leading to the next location. Note that non-determinism on discrete jumps may introduce branching, i.e., it requires the computation of more than one flowpipe. The boundedness of the analysis is usually achieved by limiting the length of a flowpipe and the number of discrete jumps.

To compute the set of reachable states  $reach_\infty^{\mathcal{H}}(\sigma)$  for unbounded time requires finding a fixpoint in the reachability analysis. For flowpipe-construction based techniques, finding fixpoints boils down to validating, whether a computed set of reachable states is fully contained in the set of previously computed state sets. As the approach accumulates over-approximation errors over time, it may happen that this statement cannot be validated [25]. In practice, researchers often check, whether the set obtained after a jump is contained in one of the already computed state sets.

**Safety verification via reachability analysis.** Reachability analysis can be used to verify safety properties by checking that the reachable states do not contain any unsafe states. A system is (bounded-)safe if  $R \cap S_{bad} = \emptyset$ , otherwise the result is inconclusive. Unbounded safety results can only be obtained in case the method is able to detect a fixpoint for all possible trajectories in all possible execution branches.

### 3.2 Temporal Specification

We use Signal Temporal Logic (STL) [5], as the temporal specification language to express the safe behavior of our controllers. Let  $\Theta$  be a set of terms of the form  $f(R)$  where  $R \subseteq S$  are subsets of variables and  $f : \mathbb{R}^{|R|} \rightarrow \mathbb{R}$  are interpreted functions. The syntax of STL is given by the grammar<sup>4</sup>

$$\varphi ::= \mathbf{true} \mid f(R) > k \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \mathbf{U}_I \varphi_2,$$

where  $f(R)$  are terms in  $\Theta$ ,  $k$  is a constant in  $\mathbb{Q}$  and  $I$  are intervals with bounds that are constants in  $\mathbb{Q} \cup \{\infty\}$ . We omit  $I$  when  $I = [0, \infty)$ . From the basic definition of STL, we can derive other standard operators as usual: conjunction  $\varphi_1 \wedge \varphi_2$ , implication  $\varphi_1 \rightarrow \varphi_2$ , eventually  $\mathbf{F}_I \varphi$  and always  $\mathbf{G}_I \varphi$ .

In our approach, we use STL specifications to handle properties beyond simple invariants. More specifically, we support the following subset of STL specifications: *invariance*  $\mathbf{G}(\varphi)$ , *bounded reachability*  $\mathbf{F}_{[0,t]}(\varphi)$ , and *bounded liveness*  $\mathbf{G}(\psi \rightarrow \mathbf{F}_{[0,t]}(\varphi))$  where  $\varphi$  and  $\psi$  are predicates over state variables and  $t$  is a time bound. We also allow assumptions about the environment such as the bounds on input variables.

STL specifications can be translated to hybrid automaton monitors. The translation is inspired by the templates used by Frehse et al. [26]. We adapt

<sup>4</sup> The STL semantics is standard and can be found in Appendix A.



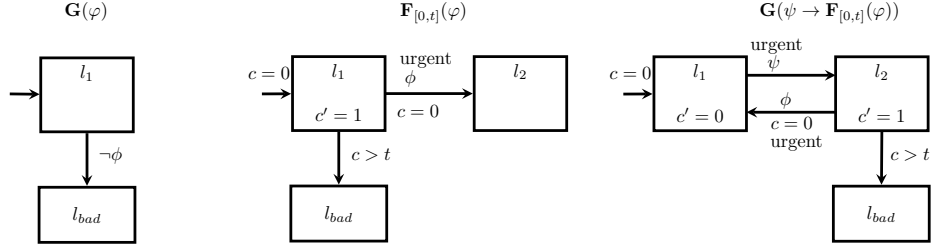


Fig. 4: Hybrid automata templates for the STL properties: *invariance* ( $\mathbf{G}(\varphi)$ ), *bounded reachability* ( $\mathbf{F}_{[0,t]}(\varphi)$ ), and *bounded liveness* ( $\mathbf{G}(\psi \rightarrow \mathbf{F}_{[0,t]}(\varphi))$ ).

the original construction to facilitate fixpoint detection, by creating (mostly) deterministic monitors instead of universal ones. Figure 4 depicts the specification automata for the STL fragments considered in this work. A specification is violated when the *sink* location  $\ell_{bad}$  is reached. Urgent transitions are encoded with location invariants and transition guards, such that no time may pass when an urgent transition is enabled. This is possible because our  $\varphi$  and  $\psi$  are half-plane constraints and we use the inverted guards as invariants. Conjunction of *invariance*, *bounded reachability*, and *bounded liveness* properties is enabled by parallel composition of monitor automata.

## 4 Verification of Adaptive Simplex Architectures

This section describes our workflow for obtaining a fully verified and adaptive Simplex architecture that is able to enforce the safety of a black box controller for temporal specifications. In this section, we first describe the setting and introduce the terminology in Section 4.1. Next, in Section 4.2, we describe how to verify a conservative known recoverable region and the techniques required to analyze systems for infinite time. Finally, in Section 4.3 we describe how to execute the Simplex architecture for temporal specifications and stateful baseline controllers, as well as how to incrementally extend the known recoverable region using proofs on demand.

### 4.1 Setting

We assume that we have a model of the plant, the baseline controller, and a bounded liveness specification. The plant model is given as a hybrid automaton  $\mathcal{H}_P$  with variables  $Var_{\mathcal{H}_P}$  and locations  $Loc_{\mathcal{H}_P}$ . We designate a subset  $U \subseteq Var_{\mathcal{H}_P}$  of the variables as controller inputs with have constant dynamics and we use variables  $X = Var_{\mathcal{H}_P}/U$  for the observable state of  $\mathcal{H}_P$ . We assume that the plant and its model  $\mathcal{H}_P$  are deterministic, i.e., that for every observable variable valuation of the plant  $X$  there exists *exactly one* location  $\ell$  in the model where  $X \in Inv_{\mathcal{H}_P}$ .

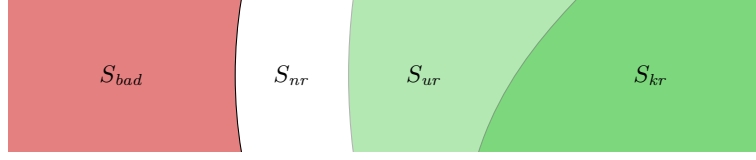


Fig. 5: Partitioning of the state space induced by a Simplex architecture.

The baseline controller is a hybrid automaton  $\mathcal{H}_{BC}$  equipped with a clock  $t$  that monitors the cycle time.  $\mathcal{H}_{BC}$  is composed with  $\mathcal{H}_P$  and can read the state  $(\ell, \nu_X) \in Loc_{\mathcal{H}_P} \times \mathbb{R}^{|X|}$  of  $\mathcal{H}_P$ . At the end of each control cycle ( $t = \delta$ ),  $\mathcal{H}_{BC}$  sets the value of  $U$  via resets on synchronized jumps. An example of the two components is depicted in Figure 2.

The specification of the system is given as an signal temporal logic (STL) formula  $\Phi$  that is converted to a hybrid automaton  $\mathcal{H}_\Phi$  as described in Section 3.2. The advanced controller is a black box that accesses the observable state  $(\ell, \nu_X)$  of the plant at the end of a control cycle and suggests an output, i.e., an assignment for each variable in  $U$  for the next control cycle.

Following classical terms, we partition the state space of  $\mathcal{H}_P \times \mathcal{H}_{BC} \times \mathcal{H}_\Phi$  into the *unsafe region*  $S_{bad}$  (the specification has been violated) and the *safe region*  $S_{safe}$  (the specification holds). The region for which the baseline controller satisfies the specification is referred to as the *recoverable region*  $S_r \subseteq S_{safe}$ , its complement is called the *non-recoverable region*  $S_{nr}$ .

The baseline controller is first verified for a small set of initial states. From verification, we obtain a subset of the recoverable region. In the following, we use the term *known recoverable region*  $S_{kr}$  for the states for which trust has been proven; the remainder of the recoverable region is the *unknown recoverable region*  $S_{ur}$ . The regions and their relations are illustrated in Figure 5.

## 4.2 Static Verification of the Baseline Controller

We perform a reachability analysis based on a flowpipe construction. We detect fixpoints to verify that the baseline controller satisfy  $\Phi$ . The hybrid automaton  $\mathcal{H} = \mathcal{H}_P \times \mathcal{H}_\Phi$  is the product of plant and specification, the components communicate as described before via shared variables and label synchronization. Since  $\mathcal{H}_\Phi$  defines the set of bad states, we use  $S_{bad}$  for the set of bad states both in  $\mathcal{H}$  and in  $\mathcal{H}_{BC} \times \mathcal{H}$ .

To guarantee that the system satisfies the specification for unbounded time, the reachability tool searches for fixpoints outside of the bad states, that means it checks whether  $reach_\infty^{\mathcal{H} \times \mathcal{H}_{BC}}(Init_{\mathcal{H} \times \mathcal{H}_{BC}}) \cap S_{bad} = \emptyset$ . We refer to the set of states that has been proven safe for unbounded time by  $S_{kr} \subseteq (Loc_{\mathcal{H}} \times Loc_{BC}, 2^{\mathbb{R}^{|Var_{\mathcal{H}}| + |Var_{BC}|}})$ . The set  $S_{kr}$  needs to satisfy the Recoverable Region Invariance: it cannot contain any bad states and it has to be closed under the reachability relation.

**Definition 2 (Recoverable Region Invariance).** *A set  $S$  fulfills the Recoverable Region Invariance if  $S \cap S_{bad} = \emptyset$  and  $reach_{\infty}^{\mathcal{H} \times \mathcal{H}_{BC}}(S) \subseteq S$ .*

If the reachability computation terminates with a fixpoint which does not include a bad state, Definition 2 is guaranteed.

**Fixpoint detection.** As fixpoint detection is a known problem for flowpipe-construction-based reachability analysis methods, several improvements were added to increase the robustness of the approach and thus the chances to find a fixpoint. Starting from a classical approach where a fixpoint is found whenever a novel initial set after a discrete jump is fully contained inside a previously computed state set we propose several improvements.

First, we augment the reachability analysis method with an interface to access external data sources for fixpoint detection. This lets us accumulate results over several runs and thus evolve  $S_{kr}$ . External data is stored efficiently in a tree-like structure similar to *octrees* [27] that subdivides the state space into cells for faster lookup. A minor, but effective improvement for the aforementioned data structure is to only store initial sets instead of all sets of states that are computed to save memory and speed up the lookup when searching for fixpoints.

Often, novel initial sets  $S'$  are not fully contained in a single, previously computed initial set  $S_i$  but are still contained in the union of several of those sets  $S' \subseteq \bigcup S_i$ . We extend fixpoint detection to handle this case by iteratively checking for all  $S_i$  whether  $S' = S'/S_i$  eventually becomes empty. Note that this check requires computing *set-difference*, which is hard for arbitrary state sets, e.g., convex polytopes, as the potentially non-convex result needs to be convexified afterwards. To overcome this, we fix our method to operate on boxes, which allows more efficient implementation of the set-difference operation.

Furthermore, in some cases, we could not find fixpoints due to *Zeno-behavior*, i.e., infinitely many discrete jumps in zero time. An example of such behavior can often be observed in *switched systems*, where the state space is partitioned into cells where the dynamics in each cell is described by a single location. For instance, having two neighboring locations  $\ell, \ell'$  connected by jumps with guards  $x \geq 5$  and  $x < 5$ , for  $x = 5$  the system can switch infinitely often between those locations without making any progress. To overcome this problem we have added detection for those Zeno-cycles that do not allow progress into our analysis method, such that these cycles get executed only once and thus can be declared a fixpoint. In contrast to the aforementioned approach to finding fixpoints, which operates on the computed state sets, this method analyzes cycles symbolically and does not cause over-approximation errors.

**Parallel composition.** Computation of  $S_{kr}$  is performed on the product of the plant-automaton, the specification-automaton, and the baseline controller automaton. To improve scalability, we feature an on-the-fly parallel composition that unrolls the product automaton during analysis as required. This improves execution speed and reduces the memory footprint.

**Algorithm 1** Execution using a Simplex architecture.

---

```

1:  $((\ell, \ell_B, \ell_\Phi), (X, U_B, X_\Phi)) \leftarrow \text{Init}_{H \times H_{BC}}$ 
2: loop
3:    $U_A \leftarrow AC(X)$ 
4:    $U_B \leftarrow BC(X)$ 
5:    $((\ell', \ell'_B, \ell'_\Phi), (X', U'_B, X'_\Phi)) \leftarrow \text{reach}_{=\delta}^{H \times H_{BC}}(((\ell, \ell_B, \ell_\Phi), (X, U_A, X_\Phi)))$ 
6:    $\text{suc} \leftarrow ((\ell', \ell'_B, \ell'_\Phi), (X', U'_B, X'_\Phi)) \subseteq S_{kr}$ 
7:   if  $\neg \text{suc}$  then
8:      $(\text{suc}, S_{kr}) \leftarrow \text{TRAIN}(S_{kr}, ((\ell', \ell'_B, \ell'_\Phi), (X', U'_B, X'_\Phi)))$ 
9:   end if
10:  if  $\text{suc}$  then
11:     $X \leftarrow \text{runPlant}(U_A, \delta)$ 
12:  else
13:     $((\ell', \ell'_B, \ell'_\Phi), (X', U_B, X'_\Phi)) \leftarrow \text{reach}_{=\delta}^{H \times H_{BC}}(((\ell, \ell_B, \ell_\Phi), (X, U_B, X_\Phi)))$ 
14:     $X \leftarrow \text{runPlant}(U_B, \delta)$ 
15:  end if
16:   $(\ell, \ell_B, \ell_\Phi, X_\Phi) \leftarrow (\ell', \ell'_B, \ell'_\Phi, X'_\Phi)$   $\triangleright$  Update for next loop iteration
17: end loop
18: procedure  $\text{TRAIN}(S_{kr}, \sigma)$ 
19:    $S_{new} \leftarrow \text{bloat}(\sigma)$ 
20:   if  $\text{reach}_{\infty}^{H \times H_{BC}}(S_{new}, S_{kr}) \cap S_{bad} = \emptyset$  then
21:     return  $(\top, S_{kr} \cup \text{reach}_{\infty}^{H \times H_{BC}}(S_{new}, S_{kr}))$ 
22:   else
23:     return  $(\perp, S_{kr})$ 
24:   end if
25: end procedure

```

---

**4.3 Simplex Execution with Proofs on Demand**

Once a known recoverable region  $S_{kr}$  has been verified it can be used in a switching logic. The intuition is to analyze the predicted set of reachable states for the plant (and the specification) *ahead* for one control cycle and decide whether to use the advanced controller or the baseline controller. The decision is based on whether these results are compatible with the previously computed recoverable region  $S_{kr}$ , or if  $S_{kr}$  can be extended to allow for these states. In the following, we give a more technical description of this approach which is also shown in Algorithm 1.

The initial state is obtained from the *model* of the composition of plant and specification. In a loop, the method receives the suggested outputs  $U_A, U_B$  from both advanced and baseline controller (Lines 3 and 4) based on the current observable state  $X$ . Since advanced controller and baseline controller may be stateful, this step may also update their internal states based on  $(\ell, X)$  during the computation of the controller output. In a next step, we use reachability analysis from the current state  $((\ell, \ell_B, \ell_\Phi), (X, U_A, X_\Phi))$  to obtain all possible  $\delta$ -reachable states  $((\ell', \ell'_B, \ell'_\Phi), (X', U'_B, X'_\Phi))$  of the plant, the baseline controller, and the specification when using the output from the advanced controller (Line 5).

The analysis is done for the length  $\delta$  of one control cycle. Note that in this step, we analyze the composition of the plant, the specification, and the baseline controller using the output of the advanced controller to obtain all possible initial states for the next iteration. The idea is to be able to validate, whether after having invoked the advanced controller, the resulting configuration of the plant and the specification yields a configuration from which the baseline controller can ensure safety afterwards if required.

The system in its current state is *recoverable* when using the advanced controller, if the newly obtained states  $((\ell', \ell'_B, \ell'_\Phi), (X', U'_B, X'_\Phi))$  are fully contained in  $S_{kr}$  (Line 6). The training function TRAIN can be invoked to attempt to extend  $S_{kr}$  (Line 8) if the new states are not yet included. If the new states are contained in  $S_{kr}$  (possibly after extending it), the plant will be run for one control cycle with the control input of the advanced controller (Line 11). Otherwise, the plant is executed using the baseline controller output (Line 14). In both cases, the state of the plant is observed and stored in  $X$ .

The procedure  $\text{TRAIN}(S_{kr}, \sigma)$  (Line 18) checks, if the new observation is safe for unbounded time. In this case, it returns the new recoverable region together with a Boolean flag  $\top$ . Otherwise, the procedure returns  $\perp$  and the old recoverable region. The construction of the specification automaton ensures that a bad state can never be left, if any point between now and  $\sigma$  would unsafe so is  $\sigma$ . To produce results that generalize beyond a single state we *bloat* the state set by extending it in all dimensions to a configurable size. Unbounded time safety is checked for this enlarged set. This can be established either by proving that all trajectories reach  $S_{kr}$  or by finding a new fixpoint. Extending  $S_{kr}$  using TRAIN preserves recoverability (Definition 2).

**Proposition 1.** *Let  $S'_{kr}$  be the set computed by  $\text{TRAIN}(S_{kr}, s)$  then*

$$\begin{aligned} (S_{kr} \cap S_{bad} = \emptyset \wedge \text{reach}_{\infty}^{H \times H_{BC}}(S_{kr}) \subseteq S_{kr}) \\ \Rightarrow (S'_{kr} \cap S_{bad} = \emptyset \wedge \text{reach}_{\infty}^{H \times H_{BC}}(S'_{kr}) \subseteq S'_{kr}) \end{aligned}$$

The intersection of the states added by TRAIN and of  $S_{kr}$  with  $S_{bad}$  is empty. Thus, this also holds for  $S'_{kr}$ . There are two cases to show that the evolution of all states remains in  $S'_{kr}$ : First, the added states originate from a flowpipe that fully leads into  $S_{kr}$ . In this case, all added states will have a trajectory into  $S_{kr}$  when using the baseline controller, where they will stay by the assumption for  $S_{kr}$ . In the second case the new recoverable region that is added that has its own fixpoint, i.e., is safe for unbounded time. The used reachability method guarantees that every trajectory stays in this region which is a subset of  $S'_{kr}$ . Thus  $S'_{kr}$  satisfies both properties from Definition 2, i.e., a system controlled by Algorithm 1 satisfies  $\Phi$ , if its initial state is in  $S_{kr}$ .

The evolutionary nature of this approach allows to provide *proofs on demand* even during running time, provided the system environment is equipped with enough computational power to perform reachability analysis. Since this is in general not the case, the approach can be adapted to collect potential new initial sets  $S_{AC}$  and verify those offline or run verification asynchronously (online). In

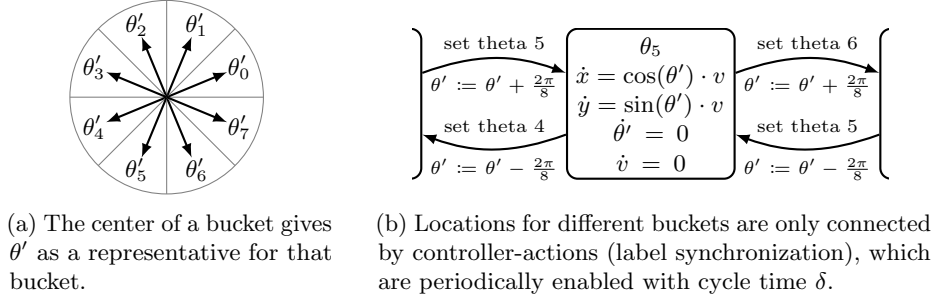


Fig. 6: Modelling approach for the discretized car for 8 buckets.

the later cases, since safety cannot directly be shown for  $S_{AC}$ , the system switches to using the baseline controller and results obtained offline or asynchronously can be integrated into future iterations.

## 5 Case Study: Autonomous Racing Car

We evaluate our approach to a controller for an autonomous racing car. The car is modeled as a point mass, observables are the position  $(x, y)$ , the heading  $(\theta)$ , and its velocity  $(v)$ . The car can be modeled by a hybrid automaton with a single location and non-linear dynamics. For simplification, we allow instantaneous changes of the velocity and do not model acceleration. To obtain a *linear* hybrid system, we discretize  $\theta$  and replace it with a representative such that the transcendental terms become constants (see Figure 6a); the number of buckets for this discretization is parameterized and induces multiple locations (one for each bucket for each discretized variable, see Figure 6b).

The car is put on different circular racetracks where the safety specification is naturally given by the track boundaries. Each track is represented as three collections of convex polygonal shapes  $P_{in}, P_{out}, P_{curbs} \subseteq \mathbb{R}^2$  which define the inner and outer boundary of the track (see yellow area in Figure 7a), as well as the curbs on the border of the track. Whenever the car enters the curbs it must exit them within 2 time units. Formally, the specification is  $\Phi = \mathbf{G}((x, y) \notin P_{in}) \wedge \mathbf{G}((x, y) \notin P_{out}) \wedge \mathbf{G}(((x, y) \in P_{curbs}) \rightarrow \mathbf{F}_{[0,2]}((x, y) \notin P_{curbs}))$ .

**Baseline Controller.** To model the baseline controller each track is subdivided into an ordered sequence of straight segments. The control inputs of the baseline controller attempt to drive the car to the center region of the current segment where the car stops. To model this behavior, each segment is subdivided into several zones with different dynamics, depending on the relative position of the zone to the center of the segment.

**Advanced Controller.** The advanced controller implements a pure pursuit controller [28] that is equipped with a set of waypoints along the track. Waypoints are either given by points in the middle of the track on the boundary between two segments or in a more advanced setup obtained by a raceline optimizer tool [29].

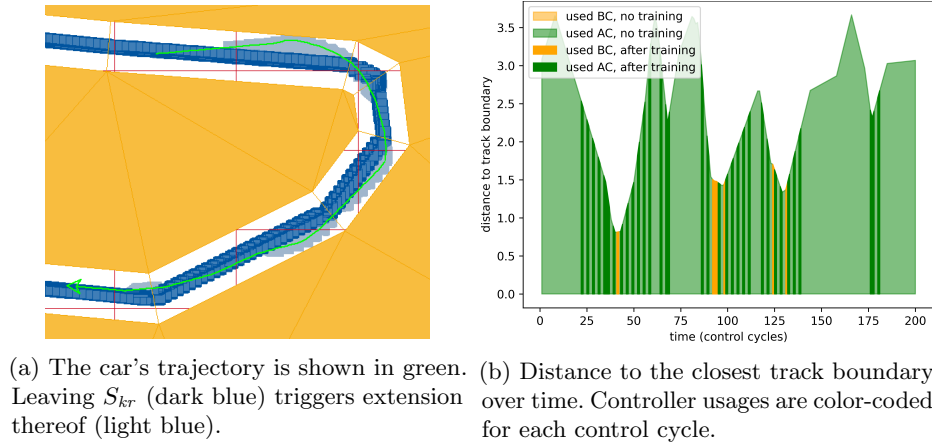


Fig. 7: Execution of the race car with online verification.

**Results and Observations.** For evaluation, we consider several tracks that are either simple toy examples such as square- or L-shaped tracks or linearizations of actual F1 racetracks.

We outline some of the results that we obtained during evaluation here, we provide a full set of images and videos online <sup>5</sup>.

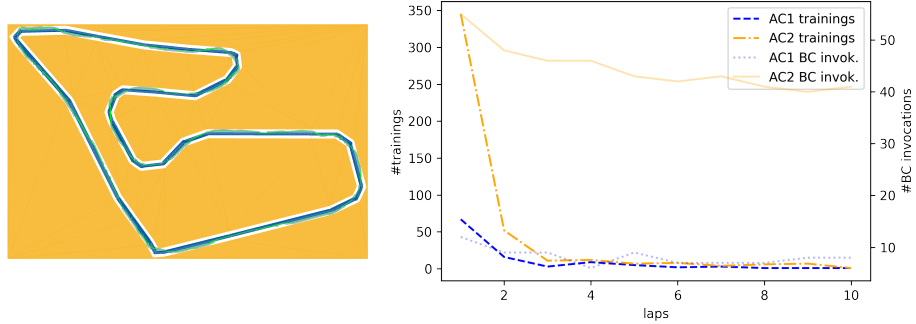
The designed baseline controller is relatively conservative, it prioritizes safety over progress as it steers the car toward the center line of the track and then stops. This behavior enables fast computation of unbounded safety results but as a drawback does not allow for large extensions of  $S_{kr}$  during a single training run. To show the influence of proofs on demand, we synthesized  $S_{kr}$  for the center area of a whole track a priori. Depending on the selection of waypoints for the advanced controller, naturally, the usefulness of the initial  $S_{kr}$  varies. Waypoints in the middle tend to induce fewer extensions of  $S_{kr}$  (or, fewer invocations of the baseline controller in case proofs on demand are disabled) than the set of waypoints generated by the raceline optimizer, which selects points allowing a path with less curvature. A visualization of the optimized trajectory,  $S_{kr}$ , as well as its extension can be seen in Figure 7a.

We recorded the number of invocations of adaptive proofs over time (see e.g. Figure 7b), the results are plotted in Figure 8 for the two sets of waypoints. We can observe, that initially many proofs are required to saturate  $S_{kr}$  in the first lap. Later laps require fewer adaptations, the number of proofs stabilizes at around 40 proofs per lap. This results from the controller not ending up perfectly in its starting position from the lap before, such that evolving  $S_{kr}$  is necessary to use the advanced controller as often as possible. The initial position used to generate proofs on demand is bloated to allow larger extensions of  $S_{kr}$  at

<sup>5</sup> <https://github.com/modass/simplex-architectures/wiki/Experimental-results>

bloating	BC invocations	extensions
0.25	123	2945
0.5	142	1329
0.75	189	1133
1.0	268	803

Table 1: Bloatings used for proofs on demand affect the success of this approach.

Fig. 8: Successful extensions of  $S_{kr}$  (left: blue) per lap for the F1 track. Starting from a moderately large known recoverable region for two sets of waypoints (standard = AC1, optimized = AC2).

a time. Our experiments with different bloatings (Table 1) show two opposing effects with increasing bloating: (1) fewer requests for extensions due to the larger extension, and (2) more invocations of the baseline controller as fewer extensions are successful due to the attempt of more aggressive expansion.

On a standard desktop computer, the computation of the initial recoverable region takes about 20 minutes, performing a proof-on-demand to extend the recoverable region takes less than a second, testing whether to switch with out extending takes about 10 milliseconds.

## 6 Conclusion

We have presented a method to incorporate proofs on demand in a fully automated Simplex architecture toolchain to ensure controllers obey a given temporal specification. Our method operates incrementally, fitting the recoverable region of the baseline controller to the behavior of the running advanced controller. Since our Simplex architecture adapts to the advanced controller, it allows performance increases by avoiding unnecessary switches to the baseline controller and invokes fewer verification queries at later stages of the execution phase.

One direction for future work is to use the robustness values of the STL specification to fine-tune the switching mechanism from the baseline controller to the advanced controller. Furthermore, we want to combine our Simplex architecture



with reinforcement learning such that the architecture guides the learning phase via reward shaping and, at the same time, ensures correctness during training.

**Acknowledgements** This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreements No 956123 (FOCETA). This preprint has not undergone peer review (when applicable) or any post-submission improvements or corrections. The Version of Record of this contribution is published in LNCS 13872, and is available online at [https://doi.org/10.1007/978-3-031-32157-3\\_8](https://doi.org/10.1007/978-3-031-32157-3_8)

## References

1. L. Sha, “Using simplicity to control complexity,” *IEEE Software*, no. 4, pp. 20–28, 2001.
2. D. Seto, B. Krogh, L. Sha, and A. Chutinan, “The simplex architecture for safe online control system upgrades,” in *ACC*. IEEE, 1998, pp. 3504–3508.
3. T. L. Crenshaw, E. L. Gunter, C. L. Robinson, L. Sha, and P. R. Kumar, “The simplex reference model: Limiting fault-propagation due to unreliable components in cyber-physical system architectures,” in *RTSS*. IEEE Computer Society, 2007, pp. 400–412.
4. R. Alur, C. Courcoubetis, T. A. Henzinger, and P.-H. Ho, “Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems,” in *Hybrid systems*. Springer, 1992, pp. 209–229.
5. O. Maler and D. Nickovic, “Monitoring temporal properties of continuous signals,” in *FORMATS/FTRTFT*, ser. Lecture Notes in Computer Science, vol. 3253. Springer, 2004, pp. 152–166.
6. S. Schupp, E. Ábrahám, I. B. Makhoul, and S. Kowalewski, “HyPro: A C++ library of state set representations for hybrid systems reachability analysis,” in *NFM*, ser. Lecture Notes in Computer Science, vol. 10227, 2017, pp. 288–294.
7. D. T. Phan, R. Grosu, N. Jansen, N. Paoletti, S. A. Smolka, and S. D. Stoller, “Neural simplex architecture,” in *NFM*, ser. Lecture Notes in Computer Science, vol. 12229. Springer, 2020, pp. 97–114.
8. T. B. Ionescu, “Adaptive simplex architecture for safe, real-time robot path planning,” *Sensors*, vol. 21, no. 8, 2021.
9. S. Shivakumar, H. Torfah, A. Desai, and S. A. Seshia, “SOTER on ROS: A run-time assurance framework on the robot operating system,” in *RV*, ser. Lecture Notes in Computer Science, vol. 12399. Springer, 2020, pp. 184–194.
10. J. Yang, M. A. Islam, A. Murthy, S. A. Smolka, and S. D. Stoller, “A simplex architecture for hybrid systems using barrier certificates,” in *SAFEComp*, ser. Lecture Notes in Computer Science, vol. 10488. Springer, 2017, pp. 117–131.
11. U. Mehmood, S. D. Stoller, R. Grosu, and S. A. Smolka, “Collision-free 3d flocking using the distributed simplex architecture,” in *Formal Methods in Outer Space*, ser. Lecture Notes in Computer Science, vol. 13065. Springer, 2021, pp. 147–156.
12. U. Mehmood, S. D. Stoller, R. Grosu, S. Roy, A. Damare, and S. A. Smolka, “A distributed simplex architecture for multi-agent systems,” in *SETTA*, ser. Lecture Notes in Computer Science, vol. 13071. Springer, 2021, pp. 239–257.
13. S. Bak, K. Manamcheri, S. Mitra, and M. Caccamo, “Sandboxing controllers for cyber-physical systems,” in *ICCPs*. IEEE Computer Society, 2011, pp. 3–12.

14. T. T. Johnson, S. Bak, M. Caccamo, and L. Sha, “Real-time reachability for verified simplex design,” *ACM Transactions on Embedded Computing Systems*, vol. 15, no. 2, pp. 26:1–26:27, 2016.
15. D. Marta, C. Pek, G. I. Melsión, J. Tumova, and I. Leite, “Human-feedback shield synthesis for perceived safety in deep reinforcement learning,” *IEEE Robotics Autom. Lett.*, vol. 7, no. 1, pp. 406–413, 2022.
16. T. D. Simão, N. Jansen, and M. T. J. Spaan, “Alwayssafe: Reinforcement learning without safety constraint violations during training,” in *AAMAS ’21: 20th International Conference on Autonomous Agents and Multiagent Systems, Virtual Event, United Kingdom, May 3-7, 2021*, F. Dignum, A. Lomuscio, U. Endriss, and A. Nowé, Eds. ACM, 2021, pp. 1226–1235.
17. M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu, “Safe reinforcement learning via shielding,” in *AAAI*. AAAI Press, 2018, pp. 2669–2678.
18. M. Z. Romdlony and B. Jayawardhana, “Stabilization with guaranteed safety using control lyapunov-barrier function,” *Automatica*, vol. 66, pp. 39–47, 2016.
19. S. Prajna and A. Jadbabaie, “Safety verification of hybrid systems using barrier certificates,” in *HSCC*, ser. Lecture Notes in Computer Science, vol. 2993. Springer, 2004, pp. 477–492.
20. C. Belta, B. Yordanov, and E. Aydin Gol, “Discrete-time dynamical systems,” in *Formal Methods for Discrete-Time Dynamical Systems*. Springer, 2017.
21. T. A. Henzinger, “The theory of hybrid automata,” in *Verification of digital and hybrid systems*. Springer, 2000, pp. 265–292.
22. T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya, “What’s decidable about hybrid automata?” *Journal of Computer and System Sciences*, vol. 57, no. 1, pp. 94–124, 1998.
23. S. Schupp, “State set representations and their usage in the reachability analysis of hybrid systems,” Ph.D. dissertation, RWTH Aachen University, Aachen, 2019.
24. A. Chutinan and B. H. Krogh, “Computational techniques for hybrid system verification,” *IEEE transactions on automatic control*, vol. 48, no. 1, pp. 64–75, 2003.
25. S. Schupp, E. Ábrahám, X. Chen, I. B. Makhoul, G. Frehse, S. Sankaranarayanan, and S. Kowalewski, “Current challenges in the verification of hybrid systems,” in *International Workshop on Design, Modeling, and Evaluation of Cyber Physical Systems*. Springer, 2015, pp. 8–24.
26. G. Frehse, N. Kekatos, D. Nickovic, J. Oehlerking, S. Schuler, A. Walsch, and M. Woehrle, “A toolchain for verifying safety properties of hybrid automata via pattern templates,” in *ACC*. IEEE, 2018, pp. 2384–2391.
27. D. Meagher, “Geometric modeling using octree encoding,” *Computer Graphics and Image Processing*, vol. 19, no. 2, pp. 129–147, 1982.
28. O. Amidi and C. E. Thorpe, “Integrated mobile robot control,” in *Mobile Robots V*, vol. 1388, International Society for Optics and Photonics. SPIE, 1991, pp. 504 – 523.
29. A. Heilmeier, A. Wischnewski, L. Hermansdorfer, J. Betz, M. Lienkamp, and B. Lohmann, “Minimum curvature trajectory planning and control for an autonomous race car,” *Vehicle System Dynamics*, vol. 58, no. 10, pp. 1497–1527, 2020.

## A Temporal Specification

In this section, we present signal temporal logic (STL) [5], the temporal specification language that we use to express the intended and correct behavior of our controllers.

Let  $\Theta$  be a set of terms of the form  $f(R)$  where  $R \subseteq S$  are subsets of variables and  $f : \mathbb{R}^{|R|} \rightarrow \mathbb{R}$  are interpreted functions. The syntax of STL is given by the grammar

$$\varphi ::= \mathbf{true} \mid f(R) > k \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \mathbf{U}_I \varphi_2,$$

where  $f(R)$  are terms in  $\Theta$ ,  $k$  is a constant in  $\mathbb{Q}$  and  $I$  are intervals with bounds that are constants in  $\mathbb{Q} \cup \{\infty\}$ . As customary the timing interval  $I$  may be omitted when  $I = [0, \infty)$ .

With respect to a signal  $w$ , the semantics of an STL formula is described via the satisfiability relation  $(w, t) \models \varphi$ , indicating that the signal  $w$  satisfies  $\varphi$  at the time  $t$ :

$$\begin{aligned} (w, t) &\models \mathbf{true} \\ (w, t) &\models f(R) > 0 & \text{iff} & f(w_R[t]) > 0 \\ (w, t) &\models \neg\varphi & \text{iff} & (w, t) \not\models \varphi \\ (w, t) &\models \varphi_1 \vee \varphi_2 & \text{iff} & (w, t) \models \varphi_1 \text{ or } (w, t) \models \varphi_2 \\ (w, t) &\models \varphi_1 \mathbf{U}_I \varphi_2 & \text{iff} & \exists t' \in t \oplus I, (w, t') \models \varphi_2 \text{ and} \\ & & & \forall t'' \in (t, t'), (w, t'') \models \varphi_1. \end{aligned}$$

Here we use the symbol  $\oplus$  to denote the Minkowski sum between  $t$  and  $I$ , defined as follows:  $t \oplus I = \{t + a \mid a \in I\}$ . We write  $w \models \varphi$  when  $(w, 0) \models \varphi$ .

We use  $\mathbf{U}$  as syntactic sugar for the *untimed* variants of the *until*  $\mathbf{U}_{(0, \infty)}$  operator. From the basic definition of STL, we can derive other standard operators.

tautology	<b>true</b>	$= p \vee \neg p$
contradiction	<b>false</b>	$= \neg \mathbf{true}$
conjunction	$\varphi_1 \wedge \varphi_2$	$= \neg(\neg\varphi_1 \vee \neg\varphi_2)$
implication	$\varphi_1 \rightarrow \varphi_2$	$= \neg\varphi_1 \vee \varphi_2$
eventually, finally	$\mathbf{F}_I \varphi$	$= \mathbf{true} \mathbf{U}_I \varphi$
always, globally	$\mathbf{G}_I \varphi$	$= \neg \mathbf{F}_I \neg \varphi$

In our approach we use STL specifications to handle properties beyond simple invariants. More specifically, we support the following subset of STL specifications:

<b>Invariance</b>	$\mathbf{G}(\varphi)$
<b>Bounded Reachability</b>	$\mathbf{F}_{[0, t]}(\varphi)$
<b>Bounded Liveness</b>	$\mathbf{G}(\psi \rightarrow \mathbf{F}_{[0, t]}(\varphi))$

where  $\varphi$  and  $\psi$  are predicates over state variables and  $t$  is a time bound.

Additionally, we might need some assumptions on the environment inputs. In the simplest case these will be given as intervals for all input variables.

STL specifications can be translated to hybrid automaton monitors. The translation is inspired by the templates used by Frehse et al. [26]. We adapt

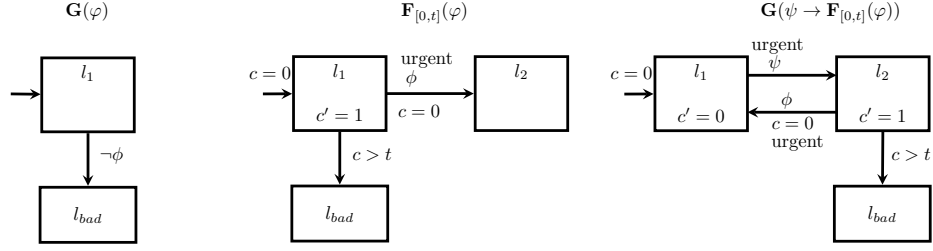


Fig. 9: Hybrid automata templates for the STL properties: *invariance* ( $\mathbf{G}(\varphi)$ ), *bounded reachability* ( $\mathbf{F}_{[0,t]}(\varphi)$ ), and *bounded liveness* ( $\mathbf{G}(\psi \rightarrow \mathbf{F}_{[0,t]}(\varphi))$ ).

the original construction to facilitate fix-point detection, by creating (mostly) deterministic monitors instead of universal ones. Figure 4 depicts the specification automata for the STL fragments considered in this work. A specification is violated when the *sink* location  $\ell_{bad}$  is reached. Urgent transitions are encoded with location invariants and transition guards, such that no time may pass when an urgent transition is enabled. This is possible because our  $\varphi$  and  $\psi$  are half-plane constraints and we use the inverted guards as invariants.

Specifications can also be expressed as a conjunction of the *invariance*, *bounded reachability*, and *bounded liveness* properties. In this case a monitor is created for every subproperty. These are then combined to one specification monitor using shared state composition. Note that each of the subproperty monitors needs to use fresh names for its locations and newly introduced state variables (the clocks  $c$ ). The locations  $\ell_{bad}$  are combined into one single bad location.