International Journal on Software Tools for Technology Transfer
https://doi.org/10.1007/s10009-025-00779-0

GENERAL

Special Issue: SPIN 2022-2024

An adaptive, provable correct simplex architecture

Benedikt Maderbacher[1] · Stefan Schupp[2] · Ezio Bartocci[2] · Roderick Bloem[1] · Dejan Ničković[3] · Bettina Könighofer[1]

Accepted: 14 January 2025
© The Author(s) 2025

Abstract

Simplex architectures optimize performance and safety by switching between an advanced controller and a base controller. We propose an approach to synthesize the switching logic and extensions of the base controller in the Simplex architectures to achieve high performance and provable correctness for a rich class of temporal specifications by maximizing the time the advanced controller is active. We achieve provable correctness by performing static verification of the baseline controller. The result of this verification is a set of states that is proven to be safe, called the recoverable region. We employ proofs on demand to ensure that the base controller is safe in those states that are visited during runtime, which depends on the advanced controller. Verification of hybrid systems is often overly conservative, resulting in smaller 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. In case the recoverable region cannot be extended using the baseline controller, we employ a repair procedure. This tries to synthesize a patch for the baseline controller and can further extend the recoverable region. Our offline and online verification relies upon reachability analysis since it allows observation-based extension of the known recoverable region. We implemented our methodology on top of the state-of-the-art tool HyPro which allowed us to automatically synthesize verified and performant Simplex architectures for advanced case studies, like safe autonomous driving on a race track.

Keywords Simplex architecture · Hybrid systems · Monitoring · Runtime enforcement · Repair

1 Introduction

Modern control applications are increasingly becoming autonomous and are gaining complexity. The adoption of the DevOps practices by the cyber-physical systems (CPSs) community enables control systems to evolve and progressively improve their performance after their deployment, by collecting and analyzing data during system operation.

The design of advanced control systems that can be trusted is a major challenge in the development of safety-critical applications. Formal verification of sophisticated controllers at the design time is typically intractable due to their inherent complexity. Therefore, runtime assurance techniques [27] that provide safety guarantees during system operation by monitoring and altering the execution of the controller are becoming an increasingly popular alternative.

Simplex architecture. The Simplex architecture [7, 26] is a popular runtime assurance framework, originally introduced to enable safe upgrades of control systems in operation. Given a *plant* P representing the physical system that may exhibit mixed discrete-continuous behavior, and a *safety*

This paper is an extended version of the "Provable Correct and Adaptive Simplex Architecture for Bounded-Liveness Properties" manuscript published at SPIN 2023 [14].

✉ B. Maderbacher
benedikt.maderbacher@tugraz.at

S. Schupp
stefan.schupp@tuwien.ac.at

E. Bartocci
ezio.bartocci@tuwien.ac.at

R. Bloem
roderick.bloem@tugraz.at

D. Ničković
dejan.nickovic@ait.ac.at

B. Könighofer
bettina.koenighofer@tugraz.at

[1] Graz University of Technology, Graz, Austria

[2] TU Wien, Vienna, Austria

[3] AIT Austrian Institute of Technology, Vienna, Austria

Published online: 04 February 2025

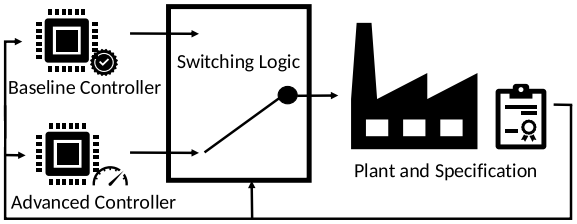


Fig. 1 Schematic of a Simplex architecture

specification φ, the Simplex architecture ensures that P satisfies the specification φ. It consists of three components, as shown in Fig. 1:

1. The *baseline controller* (BC), a simple controller that is formally verified. More specifically, the baseline controller is guaranteed to provide control inputs to the plant, which result in behaviors that satisfy φ. This holds under the assumption that P is initially in a state from which the baseline controller can satisfy φ. We call the set of such states the *recoverable region*.
2. The *advanced controller* (AC), a highly efficient, sophisticated, and possibly data-driven controller that might incorporate deep neural networks. Due to its complexity, the advanced controller is not amenable to formal verification, and its behavior does not have guarantees to always satisfy φ.
3. The *switching logic* (SL) monitors the operation of the advanced controller. Whenever the plant is about to leave the recoverable region, the switching logic hands the control to the baseline controller that ensures the satisfaction of the safety guarantee φ.

To summarize, the Simplex architecture facilitates *high performance* of the control system while also *guarantying safety* of the application. The high performance is achieved by allowing the advanced controller to have the maximal freedom and only restricting its operation to the recoverable region. The formally verified baseline controller provides the guarantee that the plant never leaves the recoverable region.

Challenges. The simple concept of a Simplex architecture is nevertheless complex to implement. The first challenge is to formally verify the baseline controller and hence provide the safety guarantees. Baseline controllers can be modeled as hybrid automata [2] that allow accurate description of systems that combine discrete computational and continuous processes. The verification of hybrid automata is challenging and exhaustive verification tools have inherent limitations in terms of scalability and accuracy. To obviate formal verification, assumptions are often made about the correctness of a given baseline controller. However, all safety guarantees meant to be provided by the Simplex architecture are lost by avoiding the formal verification step.

The second challenge is to implement a Simplex architecture that has high performance while being safe at the same time. The accumulation of overapproximation error during verification of the baseline controller typically yields an overconservative recoverable region. This can result in the switching logic giving control to the baseline controller more frequently than necessary, causing unnecessary drops in performance.

Problem statement. We start with (1) a plant and a baseline controller, both modeled as hybrid automata, (2) a safety specification expressed in the bounded-liveness fragment of signal temporal logic (STL) [15], and (3) an advanced controller. The problem is to synthesize a Simplex architecture that has high performance while being provably correct for the given STL specification.

Our approach. We first compute at design time an initial recoverable region is the form of the set of states that can be reached by the baseline controller from an initial region. We use a fixpoint detection on this recoverable region to prove its safety with respect to the specification.

We then incrementally optimize the performance of the Simplex architecture during the system operation. Whenever the advanced controller proposes an output that would result in the plant leaving the recoverable region, we perform the following steps: (1) we give control to the baseline controller, and (2) we attempt to either enlarge the recoverable region or repair the baseline controller. To enlarge the recoverable region, we first identify *suspicious states*, states that would have been reached under the execution of the advanced controller's last command. Next, we analyze how the baseline controller behaves from the suspicious states. If we prove that the baseline controller is safe from a suspicious state, we add this state and all states reachable from it to the recoverable region. Next time the advanced controller proposes a command that results in such a state, the switching logic does not interfere. If we cannot prove that the baseline controller is safe from that suspicious state, we perform a *repair* that attempts to alter the controller's behavior by attempting to synthesize a *patch*, a finite sequence of commands that steer the system back into the known recoverable region.

Contributions. This paper expands and improves the text of our preliminary paper [14] published at SPIN 2023 with the following new contributions:

- We propose a repair mechanism for the baseline controller. Whenever the Simplex architecture encounters a suspicious state that is outside the recoverable region and could not be added to it, the repair procedure attempts to create a patch consisting of a sequence of actions that bring back the system to the recoverable region.
- We extend the autonomous racing car case study with an additional track and associated experiments that further demonstrate the value of our approach.
- We add a simple illustrative example that we use throughout the paper to explain the different steps of our approach and provide intuition to the reader.

1.1 Related work

Original works on the Simplex architecture [7, 26, 27], as well as many recent works [12, 20, 28] assume to have a verified baseline controller and a correct switching logic given. Under these assumptions, the papers guarantee the 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 [18, 19, 30], and using forward or backward reachability analysis [4]. 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 overapproximate 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 overly-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-performing Simplex architectures. Similar to our approach, the work in [13] 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 [13] 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 them.

Runtime assurance covers a wide range of techniques and has several application areas, for example, enforcing safety in robotics [16] or in machine learning [29]. 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 [1]. In the continuous domain, inductive safety invariants such as a control Lyapunov functions [22] or control barrier functions [21] are used to synthesize runtime enforcers.

2 Background

2.1 Reachability analysis of hybrid systems

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

Definition 1 (Hybrid automata [2])
A hybrid automaton $\mathcal{H} = (Loc, Lab, Edge, Var, Init, Inv, Flow, Jump)$ consists of a finite set of locations $Loc = \{\ell_1, \ldots, \ell_n\}$, a finite set of labels Lab, which synchronize and coordinate state changes between automata, a finite set of jumps $Edge \subseteq Loc \times Lab \times Loc$, that allow realizing location changes, a finite set of variables $Var = \{x_1, \ldots, x_d\}$, a set of states Inv called invariant, which restricts the values ν for each location set of initial states $Init \subseteq Inv$, a flow relation $Flow$ where $Flow(\ell) \subseteq \mathbb{R}^{\dot{Var}} \times \mathbb{R}^d$, which determines for each state (ℓ, ν) the set of possible derivatives $\dot{Var}$, 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 ν' of ν.

A state $\sigma = (\ell, \nu)$ of $\mathcal{H}$ consists of a location ℓ and a valuation $\nu \in \mathbb{R}^d$ for each variable, $S = Loc \times \mathbb{R}^d$ denotes the set of all states.

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 [10] with label-synchronization on discrete jumps and shared variables.

A *path* $\pi = \sigma_1 \rightarrow_\tau \sigma_2 \rightarrow_e \rightarrow \cdots$ in $\mathcal{H}$ is an ordered sequence of states σ_i connected by *time transitions* $\rightarrow_\tau$ of length τ 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 [10] 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 π 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

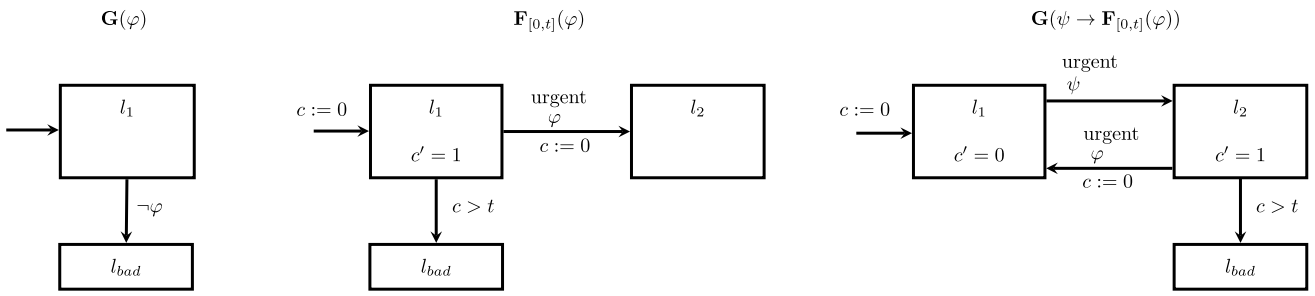


Fig. 2 Hybrid automata templates for the STL properties: *invariance* ($\mathbf{G}(\varphi)$), *bounded reachability* ($\mathbf{F}_{[0,t]}(\varphi)$), and *bounded liveness* ($\mathbf{G}(\psi \to \mathbf{F}_{[0,t]}(\varphi))$)

reachable. Since the reachability problem is in general undecidable [11], current approaches often compute overapproximations of the sets of reachable states for *bounded reachability*. Note that reachability analysis follows all execution branches, i.e., does not resolve any nondeterminism induced by discrete jumps in the model. This 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., [6]) computes a set of convex sets

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

which are guaranteed to cover all trajectories of bounded length α that are reachable from a set of states σ. We use $reach^{\mathcal{H}}_{=\alpha}(\sigma)$ to denote the set of states that are reached after exactly α time, and similarly $reach^{\mathcal{H}}_{\infty}(\sigma)$ to denote the set of states reachable for unbounded time.

The method overapproximates 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 nondeterminism 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 $reach^{\mathcal{H}}_{\infty}(\sigma)$ 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 overapproximation errors over time, it may happen that this statement cannot be validated [24]. One way is to 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.

2.2 Temporal specification

We use STL [15], as the temporal specification language to express the safe behavior of our controllers. Let Θ be a set of terms of the form $f(R)$ where $R \subseteq Var$ are subsets of variables and $f : \mathbb{R}^{|R|} \to \mathbb{R}$ are interpreted functions. The syntax of STL is given by the following grammar and we use standard semantics [15].

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

where $f(R)$ are terms in Θ, 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 $\varphi_1 \to \varphi_2$, 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* $\mathbf{G}(\psi \to \mathbf{F}_{[0,t]}(\varphi))$, where φ and ψ 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. [8]. We adapt the original construction to facilitate fixpoint detection, by creating (mostly) deterministic monitors instead of universal ones. Figure 2 depicts the specification automata for the STL fragments considered in this work. A specification is violated when the *sink* location

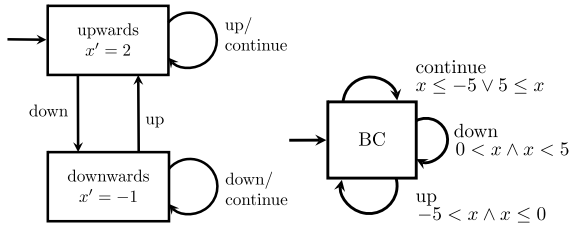


Fig. 3 Hybrid automata of illustrative example

ℓ_{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 φ and ψ are half-plane constraints and we use the inverted guards as invariants. Consider an example of $\varphi = x \leq 1$: the outgoing location uses a guard of $x \geq 1$. We assume that a state the enables φ is reached via the dynamic of the location, which is justified because our templates used resets only for clocks which are not used in guards for urgent transitions. This construction ensures that no time can pass in a location once the urgent transition is enabled because the dynamic would push the state out of the invariant. The only option available to the reachability analyzer is to follow the urgent transition. The conjunction of *invariance*, *bounded reachability*, and *bounded liveness* properties is enabled by the parallel composition of monitor automata.

3 Synthesizing adaptive simplex architectures

This section outlines our method for constructing a provably correct and adaptive Simplex architecture. In Sect. 3.1, we will first discuss the setting and the problem statement. Next, we will outline how to compute a provably correct Simplex architecture offline in Sect. 3.2. Finally, we discuss how to adapt the architecture during runtime in Sect. 3.3.

To showcase how our algorithm works, we compute a Simplex architecture for the simple hybrid automaton of the plant, as illustrated in Fig. 3(a), from the baseline controller shown in Fig. 3(b) and a simple given safety specification φ in STL. We will use this example as a running example throughout this section.

3.1 Setting and problem statement

The input of our synthesis algorithm is a model of the plant and a model of the baseline controller as well as a safety specification φ in STL. From these inputs, our algorithm computes a Simplex architecture that enforces φ, for any possible advanced controller. In the following, we discuss the individual components in detail.

3.1.1 Plant

The model of the plant is given as a hybrid automaton $\mathcal{H}_P$ with continuous variables $Var_{\mathcal{H}_P}$ and discrete locations $Loc_{\mathcal{H}_P}$. The plant $\mathcal{H}_P$ is equipped with a clock t that monitors the cycle time. At the end of each control cycle ($t = \delta$), the plant receives an input from a controller via a synchronized edges labeled with *tick*. We will refer to these communications between plant and controller at the end of a control cycle as *actions*. An action can be an assignment or reset for a designated subset of plant variables $V \subseteq Var_{\mathcal{H}_P}$, or a synchronization label from $L \subseteq Lab_{\mathcal{H}_P}$. We use U for the set of all actions (resets on V and labels L) and $X = Loc_{\mathcal{H}_P} \times \mathbb{R}^{Var_{\mathcal{H}_P}}$ for the state of $\mathcal{H}_P$.

Example
The hybrid automaton $\mathcal{H}_P$ of our illustrative example is given in Fig. 3(a). Here $\mathcal{H}_P$ has two locations (drawn as boxes) that are named `upwards` and `downwards`, and a single continuous variable x, whose flow is defined via the differential equations in the locations, e.g., $x' = -1$ in the `downwards` location. The hybrid automaton of plant ($\mathcal{H}_P$) and the baseline controller communicate via the actions (synchronization labels) `up`, `down`, and `continue`. All jumps happen at the end of a cycle, the clock t and the label *tick* are omitted for clarity.

3.1.2 Safety specification

The safety specification of the system is given as an STL formula φ. Note that the specification includes temporal safety properties like bounded liveness specifications. A safety specification φ in STL can be transformed into a hybrid automaton $\mathcal{H}_\varphi$, as discussed in Sect. 2.2.

Example
In our example, we use the safety specification

$$\varphi = \mathbf{G}\,(-3 \leq x \wedge x \leq 8)\,.$$

This specification requires that the value of x be always within the interval $[-3, 8]$.

We denote by the hybrid automaton $\mathcal{H} = \mathcal{H}_P \times \mathcal{H}_\varphi$, the product of plant automaton $\mathcal{H}_P$ and the specification automaton $\mathcal{H}_\varphi$. The components $\mathcal{H}_P$ and $\mathcal{H}_\varphi$ communicate via shared variables and label synchronization.

3.1.3 Baseline controller

The baseline controller (BC) is modeled via a hybrid automaton $\mathcal{H}_{BC}$ which is composed with $\mathcal{H}_P$ and can read the state $x \in X$ of $\mathcal{H}_P$. At the end of each control cycle (indicated by the label *tick*), $\mathcal{H}_{BC}$ selects which action to provide to the plant.

Example
The initial baseline controller $\mathcal{H}_{BC}$ is given in Fig. 3(b). This $\mathcal{H}_{BC}$ has a single location and selects its actions (*continue, down, up*) to control the plant based on the values of x.

- If $0 < x < 5$, then $\mathcal{H}_{BC}$ issues the action *down* and thereby switches the plant to the *downwards* location.
- If $-5 < x \leq 0$, then $\mathcal{H}_{BC}$ issues the action *up* and thereby switches plant to the *upwards* location.
- In all other cases, $\mathcal{H}_{BC}$ issues *continue* and lets plant continue in its current configuration.

Note that we use a slightly suboptimal baseline controller because it is well-suited to illustrate our method for computing a Simplex architecture without distracting details.

3.1.4 Advanced controller

We consider the advanced controller (AC) to be a black box. Thus, the complexity of our synthesis procedure is independent of the complexity of the advanced controller. This is particularly important if the advanced controller implements learned components. In the following, we denote a concrete advanced controller via $\mathcal{AC}$.

More formally, an advanced controller $\mathcal{AC}$ is a black box that accesses the state x of the plant $\mathcal{H}_P$ at the end of a control cycle and suggests an action from U for the next control cycle.

Example
In our illustrative example, the performance target of the used advanced controller is to keep the value of x as high as possible. Note that this performance property is conflicting with the safety specification. Thus, the advanced controller might try to increase the value of x beyond its safety limit of the value 8.

3.1.5 Problem statement

Given the hybrid automaton $\mathcal{H}$ (the product of the plant automaton $\mathcal{H}_P$ and the specification automaton $\mathcal{H}_\varphi$), and a hybrid automaton $\mathcal{H}_{BC}$ of the baseline controller, the goal is to construct a Simplex architecture such that the following two requirements hold:

- **Hard requirement – Correctness.** For any given advanced controller $\mathcal{AC}$ that can be used to control the plant, it holds that the specification φ will be satisfied, i.e., the plant will never be in an unsafe state.
- **Soft requirement – Performance.** To maximize the performance of the system, the time the advanced controller $\mathcal{AC}$ is active should be maximized.

Fig. 4 Partitioning of state space of $\mathcal{H} \times \mathcal{H}_{BC}$

3.2 Offline computation of a provable correct simplex architecture

To construct a provable correct Simplex architecture for a given baseline controller $\mathcal{H}_{BC}$, we need to synthesize a *switching logic* SL such that the safety specification φ is guaranteed to be satisfied, independent of the concrete advanced controller $\mathcal{H}_{AC}$. To achieve the correctness guarantees, we compute SL directly from $\mathcal{H}_{BC}$ performing the following steps.

Step 1 – Compute the recoverable region S_r. Following classical terms, we partition the state space of $\mathcal{H}$ into the *unsafe region* S_{bad} (the specification has been violated) and the *safe region* S_{safe} (the specification holds). To keep the notation simple, we use S_{bad} for the set of bad states both in $\mathcal{H}$ and in $\mathcal{H} \times \mathcal{H}_{BC}$ since $\mathcal{H}$ defines the set of bad states. The region for which the baseline controller $\mathcal{H}_{BC}$ satisfies the specification is referred to as the *recoverable region* $S_r \subseteq S_{safe}$. The partitioning of the state space of $\mathcal{H} \times \mathcal{H}_{BC}$ is illustrated in Fig. 4.

The recoverable region of a hybrid automaton can be computed in the following way. From a given set of initial states, we use reachability analysis to compute the *fixpoint* of all states in $\mathcal{H} \times \mathcal{H}_{BC}$ that can be reached, i.e., the fixpoint represents the set of states that will never be left by the baseline controller. If the fixpoint does not contain a state in S_{bad}, the baseline controller is provable safe, and we call the resulting set the *recoverable region* S_r of the baseline controller.

Note that if the fixpoint of $\mathcal{H} \times \mathcal{H}_{BC}$ contains an unsafe state in S_{bad}, we end the synthesis procedure since we cannot build a provable-correct Simplex architecture from a faulty baseline controller.

Step 2 – Build the switching logic SL. From S_r, we can directly build SL. To determine whether the action proposed by the advanced controller is safe, the plant model is used to simulate one control cycle.

During runtime, the switching logic monitors the current state and the potential next state of $\mathcal{H} \times \mathcal{H}_{BC}$. Let us assume that the baseline controller is in a state inside S_r, and the next selected action by the advanced controller is u_a. If all reachable states from the current state after executing u_a are in S_r, executing u_a is guaranteed to be safe and can be used. Thus, the switching logic SL gives control to the advanced controller. If executing u_a could result in a state outside of the recoverable region, the switching logic SL has to switch to the baseline controller.

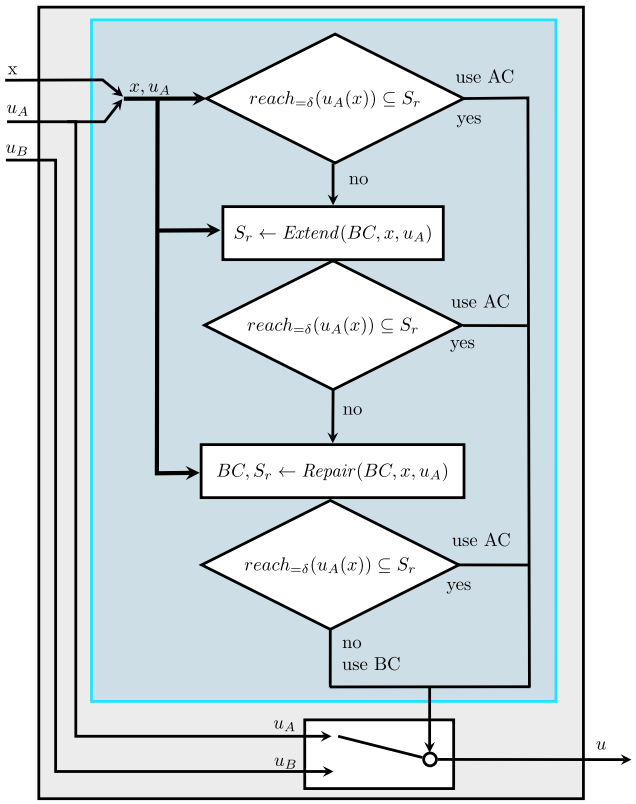


Fig. 5 Overview of our method for online adaptation of a Simplex architecture

This way, SL ensures that S_r is never left. Thus, the constructed Simplex architecture is guaranteed to be correct concerning φ.

Example
In the illustrative example, we use the set of initial states $x \in [0, 1]$. For these initial states, we obtain the recoverable region of $S_r = [-1, 2]$. The running example uses exact reachability results without overapproximation, this lets us focus on the main parts of our methodology.

The details of how to construct and use the switching logic SL are described in Sect. 4.

3.3 Online adaptation of the simplex architecture

In the previous section, we computed the set of recoverable states for the baseline controller. A switching logic that maintains operation within this recoverable region ensures system correctness.

However, this switching logic often tends to be *overly conservative*. This issue arises because precisely determining the set of reachable states for a hybrid automaton is undecidable, as indicated in [11]. Practical approaches to

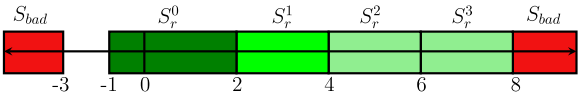


Fig. 6 Adaptations of the recoverable region in the illustrative example

reachability analysis typically result in an overestimation of the reachable states set [4]. Nonetheless, this overestimation is adequate for devising a safe switching logic. If a reliable overestimation of reachable states excludes any unsafe states, the system can be verified as safe. The degree of conservatism in the resulting switching logic directly depends on the extent of overestimation.

In this section, we will use online reachability analysis to compute a significantly less conservative switching logic that is still ensures that the safety specification φ is satisfied.

The overview of our method to adapt the Simplex architecture during runtime is illustrated in Fig. 5. Our method starts with an initial recoverable region S_r of the baseline controller $\mathcal{H}_{BC}$. At every discrete time step, the switching logic SL reads the current state x of $\mathcal{H} \times \mathcal{H}_{BC}$ (the plant and the baseline controller), and the next control actions u_a and u_b issued by the advanced controller and the baseline controller, respectively. The switching logic SL checks whether all reachable states from x after executing u_a are within S_r. If this is the case, u_a is forwarded to be executed. If a state $x' \notin S_r$ could be reached, our algorithm performs two operations to avoid an unnecessary switch to the baseline controller.

- *Extend the recoverable region.* Perform online reachability analysis to check whether x' can be added to S_r. If this is the case, all newly verified states including x' are added to S_r.
- *Repair of the baseline controller.* The state x' might be unsafe or undefined in $\mathcal{H} \times \mathcal{H}_{BC}$. The repair step aims to alter the behavior of the baseline controller in a way that x' becomes safe, while simultaneously preserving all states within the recoverable region. To do so, the repair algorithm searches for a *patch*: a patch is a finite sequence of actions that for a set of initial states steers the system back into the known recoverable region.

If neither the attempt to extend the recoverable region nor to repair the baseline controller is successful, the control output u_b is forwarded to the plant. Note that at each time step, the current recoverable region might be extended or the baseline controller may be adapted to allow more behavior. By doing so, the Simplex architecture adapts to the used advanced controller and maximizes the time the advanced controller is in control. The two adaptation techniques can be combined flexibly and depending on the setting it can also be advantageous to use only one or the other.

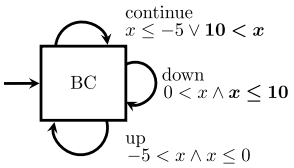


Fig. 7 BC of the illustrative example after repair

Example
The initial recoverable region of our illustrative example is $S_r^0 = [-1, 2]$. Let us assume that at time step t, $\mathcal{H} \times \mathcal{H}_{BC}$ is in the state $x_t = 2$ and the advanced controller issues the control $u_a = \texttt{up}$. In case u_a is executed, the next visited state would be $x_{t+1} = 4 \notin S_r^0$. Instead of immediately switching to the baseline controller, our algorithm first tries to extend the recoverable region. For this example, the procedure returns successfully, and the new verification results are integrated into the recoverable region extending it to $S_r^1 = [-1, 4]$. The advanced controller can now operate safely in the larger recoverable region S_r^1 resulting in a higher value of x and thus improved performance. The extension of the recoverable region is illustrated in Fig. 6.

Example
Let us continue from $S_r^1 = [-1, 4]$. Once the advanced controller tries to visit $x = 6$, extending the recoverable region is not possible. As illustrated in Fig. 3(b), once reaching $x = 6$, the baseline controller would continue in the `upwards` state until it exceeds the safety limit. Since extending S_r^1 failed, our algorithm next executes the repair procedure which tries to modify $\mathcal{H}_{BC}$ such that it is safe for the new state $x = 6$ as well. The repair procedure returns the patch `[down, down]` for $x = 6$ in the location `upwards`. Executing the patch brings the system in the state $x = 4$ in the location `downwards` which is part of the recoverable region. This patch is integrated into the existing $\mathcal{H}_{BC}$ by modifying the guards to ensure that the actions from the patch are used instead of the current behavior of $\mathcal{H}_{BC}$. Applying this repair procedure results in a recoverable region $S_r^2 = [-1, 6]$. If we apply the repair step once more, we end up in the recoverable region $S_r^3 = [-1, 8]$ and a baseline controller as depicted in Fig. 7. Figure 6 shows how the recoverable region evolves after performing the two repair steps.

We discuss the details of our approach to extend the recoverable region during runtime in Sect. 5. The details of our algorithm to repair the baseline controller are presented in Sect. 6.

4 Offline synthesis of the simplex architecture

The first step of step of our methodology is to synthesize a switching logic given a plant model and a BC. This switching logic is only required to guarantee safety for a small part of the state space, as it will be incrementally improved by our method. The main component of a switching logic is a recoverable region, i.e., a set of states for which the BC can guarantee safety for infinite time. A recoverable region S_r needs to satisfy the Recoverable Region Invariance: it cannot contain any bad states and it has to be closed under the reachability relation.

Algorithm 1 Switching Logic for a recoverable region S_r

1: $(x, x_B) \leftarrow Init_{\mathcal{H}\times\mathcal{H}_{BC}}$
2: **loop**
3: $\quad u_A \leftarrow \mathcal{AC}(x)$
4: $\quad u_B \leftarrow \mathcal{BC}(x, x_B)$
5: $\quad \left(X', X'_B\right) \leftarrow reach_{=\delta}^{\mathcal{H}\times\mathcal{H}_{BC}}(u_A(x, x_B))$
6: $\quad$ **if** $\left(X', X'_B\right) \subseteq S_r$ **then**
7: $\quad\quad x \leftarrow runPlant_\delta(u_A)$
8: $\quad$ **else**
9: $\quad\quad \left(X', X'_B\right) \leftarrow reach_{=\delta}^{\mathcal{H}\times\mathcal{H}_{BC}}(u_B(x, x_B))$
10: $\quad\quad x \leftarrow runPlant_\delta(u_B)$
11: $\quad$ **end if**
12: $\quad$ select x_B s.t. $(x, x_B) \in \left(X', X'_B\right)$
13: **end loop**

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

Once a known recoverable region S_r 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_r

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 actions 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 x during the computation of the controller output. In a next step, we use reachability analysis from the current state (x, x_B) to obtain all possible δ-reachable states $\left(X', X'_B\right)$ 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 δ 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 (X', X'_B) are fully contained in S_r (Line 6). If the new states are contained in S_r, the plant will be run for one control cycle with the control input of the advanced controller (Line 7). Otherwise, the plant is executed using the baseline controller output (Line 10). In both cases, the state of the plant is observed and stored in x and the internal state of the BC is updated.

4.1 Computing a recoverable region with fixpoint verification

A classic method to compute a set that satisfies the recoverable region invariance is to use a reachability analysis with fixpoint detection.

To guarantee that the system satisfies the specification for unbounded time, the reachability tool searches for fixpoints outside of the bad states, which 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 $S_r \subseteq Loc_{\mathcal{H}} \times Loc_{BC} \times 2^{\mathbb{R}^{|Var_{\mathcal{H}}|+|Var_{BC}|}}$.

We perform a reachability analysis based on a flowpipe construction. We detect fixpoints to verify that the baseline controller satisfies φ. If the reachability computation terminates with a fixpoint that 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 of finding 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.

Octrees. 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_r. External data is stored efficiently in a tree-like structure similar to *octrees* [17] that subdivides the state space into cells for faster lookup. For each location, we create a tree whose nodes represent a hierarchical partition of the state space, i.e., nodes of layer $i+1$ are a partition of the nodes on layer i. Computed sets are stored in the leaves of this structure to enable faster lookup; if a cell is fully covered this information can be cached for faster results. 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 nonconvex result needs to be convexified afterward. To overcome this, we fixed our method to operate on boxes, which allows more efficient implementation of the set-difference operation.

Zeno-behavior. The fixpoint detection might not terminate due to *Zeno-behavior*, i.e., infinitely many discrete jumps in zero time.

Example
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 are described by a single location. For instance, having two neighboring locations ℓ, ℓ' connected by jumps with guards $x \geq 5$ and $x \leq 5$, 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 overapproximation errors. Intuitively, for a path π leading to a reachable location ℓ, we iteratively compute sets of states that would possibly allow Zeno behavior: initially, we consider the set of states satisfying the guard condition of the incoming transition to ℓ. Going back in the considered path, we alternatingly add constraints for invariant conditions and further incoming transitions along the locations and transition on path π while also adding transformations according to the reset functions on the transitions. This way, we can encode a symbolic expression representing the set of states that enables Zeno-behavior. Checking containment of the actual state sets and the computed symbolic set allows finding Zeno-cycles of length up to the length of π.

Parallel composition. Computation of S_r 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 2 Adapting the recoverable region

```
1:  procedure EXTEND(S_r, S_AC)
2:      S_0 ← bloat(S_AC)
3:      for n ∈ [1, itermax] do
4:          S_n = reach_{=δ}^{H×H_BC}(S_{n−1})
5:          if S_n ∩ S_bad ≠ ∅ then
6:              return (⊥, S_r)
7:          else if S_n ⊆ S_r ∪ (⋃_{i∈[1,n−1]} S_i) then
8:              return (⊤, S_r ∪ (⋃_{i∈[1,n−1]} S_i)))
9:          end if
10:     end for
11:     return (⊥, S_r)        ▷ Tread as unsafe if safety cannot be established in itermax time steps.
12: end procedure
```

5 Extending the recoverable region online

In Sect. 4 we have indicated how the initial recoverable region S_r for a given baseline controller (BC) can be determined using reachability analysis. This restricts the advanced controller (AC)'s operation to a small part of the state space, even though the BC could ensure safety for a larger region. To overcome this, we extend the known recoverable region for the BC based on additional system observations.

As described in Sect. 4, in every control cycle, after having computed the next controller outputs, a simulation of the plant using the controller outputs from the AC for the next cycle is performed (see also Line 5 in Algorithm 1). This simulation may result in a set of states S_{AC} for which no safety results are available. In this case, the set of states S_{AC} can be used as an input to determine whether it can be declared safe for the baseline controller, similarly to the static analysis employed ahead of time. The procedure $\textsc{Extend}(S_r, S_{AC})$ (Algorithm 2) performs a reachability analysis from S_{AC} to determine whether S_{AC} is safe for unbounded time when using the BC. In this case, it returns the new recoverable region together with a Boolean flag $\top$. Otherwise, the procedure returns $\bot$ and the old recoverable region.

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_r or by finding a new fixpoint. It is sufficient to check if the sets S_i are safe, as the construction of the specification automaton ensures that a bad state can never be left and if there is a bad state between S_i and S_{i+1} a bad state will also be in S_{i+1}.

Extending S_r using $\textsc{Extend}$ preserves recoverability (Definition 2).

Proposition 1 (Extension preserves recoverable region invariance)
Let S_r' be the set computed by $\textsc{Extend}(S_r, s)$ then

$$(S_r \cap S_{bad} = \emptyset \wedge reach_{\infty}^{\mathcal{H}\times\mathcal{H}_{BC}}(S_r) \subseteq S_r)$$
$$\Rightarrow (S_r' \cap S_{bad} = \emptyset \wedge reach_{\infty}^{\mathcal{H}\times\mathcal{H}_{BC}}(S_r') \subseteq S_r').$$

The intersection of the states added by $\textsc{Extend}$ and of S_r with S_{bad} is empty. Thus, this also holds for S_r'. There are two cases to show that the evolution of all states remains in S_r': First, the added states originate from a flowpipe that fully leads into S_r. In this case, all added states will have a trajectory into S_r when using the baseline controller, where they will stay by the assumption for S_r. 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_r'. Thus S_r' satisfies both properties from Definition 2, i.e., a system controlled by Algorithm 1 using S_r' from Algorithm 2 satisfies φ, if its initial state is in S_r.

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 points σ and verify those offline or run verification asynchronously (online). In the later cases, since safety cannot directly be shown for σ, the system switches to using the baseline controller and results obtained offline or asynchronously can be integrated into future iterations.

6 Repair of the baseline controller

The BC can be undefined or show unsafe behavior in some states. In this case the verification will fail and the plant is not permitted to visit this state. In many cases, however,

Algorithm 3 Repair of the BC

```
1:  procedure REPAIR(BC, S_r, S_AC)
2:      S_new ← bloat(S_AC)
3:      for all p ← GENERATEPATCHCANDIDATES(S_new) do
4:          S_0, ..., S_m ← REACHPATCH(p)
5:          if ∀i : S_i ∩ S_bad = ∅ and S_m ⊆ S_r ∪ (⋃_{i∈[0,m−1]} S_i) then
6:              BC' ← APPLYPATCH(BC, p)
7:              return (⊤, BC', S_r ∪ (⋃_{i∈[0,m−1]} S_i))
8:          end if
9:      end for
10:     return (⊥, BC, S_r)
11: end procedure
```

there is a behavior for the BC that could be used instead that can guarantee safety. We propose to automatically synthesize patches for an existing BC to adapt it to work correctly for new states. This will allow the S_r region to be extended to these states and thus improve the performance of the AC.

A patch $p = (S_{start}, t)$ consists of a set of start states S_{start} and a finite sequence of actions $t \in U^m$ where m is the length of the patch sequence. For a patch to be correct, it needs to end in the known recoverable region S_r. Repairing a BC for a state requires three parts: searching patch sequences, verifying a patch, and integrating the patch in the existing controller. We will start by looking at the verification of a patch candidate.

6.1 Verifying patch candidates

To verify a patch, we translate it into a hybrid automaton and use reachability analysis. In Algorithm 3 this is done by REACHPATCH. It creates an automaton that consist of a chain of locations where each transition corresponds to one action of the patch sequence which is executed when changing locations. The automaton $\mathcal{H}_T$ generated for p has $m + 1$ locations and m edges. The locations are $Loc = \{\ell_1, \ldots, \ell_{m+1}\}$, the edges are $Edge = \{(\ell_i, \{tick\} \cup t_i, \ell_{i+1}) \mid i \in 1, \ldots, m\}$ and the jumps are $Jump((\ell_i, \{tick\} \cup t_i, \ell_{i+1})) = \{(s, t_i(s)) \mid s \in S\}$. The invariant of every location is true and there are no variables or flow. Let $S_i = proj_{\mathcal{H}}(reach^{\mathcal{H}\times\mathcal{H}_T}_{=i\cdot\delta}(S_{start}))$ be the set of states the patch p can reach after i cycles, where $proj_{\mathcal{H}}$ projects a set states from $\mathcal{H} \times \mathcal{H}_T$ to its $\mathcal{H}$ component. Given the state sets found by REACHPATCH, it can be determined if a patch is correct. None of the reached states may intersect with S_{bad}. Additionally, the patch must end in some BC state $s_{target} \in S_{BC}$ such that $S_i \times \{s_{target}\} \subseteq S_r$ for some i, or it has to include its own fixpoint, i.e., there are i and j such that $S_i \subseteq S_j$. A patch sequence ends in the known recoverable region, but it might end in a BC state that is different from its starting state.

6.2 Patch generation

The synthesis of patches relies on a way to generate patch candidates, i.e., a finite sequence of actions for a given set of start states. Different implementations can be used for this step.

To effectively enumerate candidates, the possible actions have to be discretized. This is only a minor limitation as most BC use discrete actions anyway and for others an appropriate discretization can be chosen to get patches for most correctable states.

One way is to enumerate all sequences up to a given length. This can be improved by providing domain-specific templates that limit the search space to sensible action sequences.

Another option is to use the actions suggested by the advanced controller. The plant is simulated using the AC's action for one cycle, one state is chosen out of the resulting states and the AC is queried again. Repeating this for a fixed number of control cycles results in a sequence of actions that is close to the behavior the AC would show without corrective actions. The idea behind this is that if the AC is safe and reaches the recoverable region we can copy this behavior to the BC.

6.3 Applying a patch

Once a patch has been found and verified the BC needs to be updated to the new behavior. This should strictly increase the known recoverable region, i.e., no prior proof should be invalidated. We also want to intercept longer patch sequences on the way. If a state that is part of a patch sequence is reached the verification result should be usable, even if the state is not part of the start state set of the patch.

Although the verification of the patch is done using one location per cycle, the patch can be integrated into the BC without adding new locations. For every step in the patch sequence, a new jump is added to the automaton. Let ℓ_{start} be the BC location in the start set S_{start}

of the patch. For simplicity, we assume that all states in S_{start} are from the same location, if this is not the case the process needs to be repeated for every location. The edges get extended by adding a self-loop in the start location $Edge' = Edge \cup \{(\ell_{start}, \{tick\}, \ell_{start})\}$. First, the guards of all existing transitions are restricted to exclude the newly added jumps $Jump'((\ell_{start}, \{tick\}, \ell)) = \{(s, s') \mid (s, s') \in Jump((\ell_{start}, \{tick\}, \ell)) \wedge s \notin S_n \wedge n \in 1, \ldots, m\}$. Next, the jump relation is extended to $Jump'((\ell_{start}, \{tick\}, \ell_{start})) = \{(s, t_n(s)) \mid s \in S_n \wedge n \in 1, \ldots, (m-1)\}$. The last jump might lead to a different BC location. $Jump'((\ell_{start}, \{tick\}, \ell_{target})) = \{(s, \_), (t_m(s), s_{target})) \mid s \in S_m\}$. The last jump of the patch sequence moves from ℓ_{start} to the target location ℓ_{target} that was reached in S_r and also set the internal state of the BC to match the state s_{target}.

Proposition 2 (Repair preserves verified states and recoverable region invariance)
Let BC' and S'_r be the controller and recoverable region computed by $\textsc{Repair}(BC, S_r, s)$ then $S_r \subseteq S'_r$ and

$$(S_r \cap S_{bad} = \emptyset \wedge reach_{\infty}^{\mathcal{H} \times \mathcal{H}_{BC}}(S_r) \subseteq S_r)$$
$$\Rightarrow (S'_r \cap S_{bad} = \emptyset \wedge reach_{\infty}^{\mathcal{H} \times \mathcal{H}_{BC'}}(S'_r) \subseteq S'_r).$$

The Repair function modifies S_r by adding additional states ensuring $S_r \subseteq S'_r$. Verifying the patch candidate using reachability analysis guarantees that none of the added states are in S_{bad}, therefore $S'_r \cap S_{bad} = \emptyset$ holds. For any state in S'_r, BC' defines an action as either the action in BC or the action defined in the patch p. The proof for recoverable region invariance is via induction on control cycles. Every state in S_r reaches a state in S_r at the end of the control cycle which is also in S'_r because $S_r \subseteq S'_r$. Every state added by the patch p reaches a state that is either in another set added by p or in S_r, which are all part of S'_r. Thus by induction, any state in S'_r remains in S'_r for infinite time when following BC'.

7 Experimental evaluation

We implemented a prototype of our method in C++ using the hybrid automata reachability analysis tool HyPro [25] to perform the reachability queries. Two case studies were used to evaluate our method, a coupled water tank system and an autonomous racing car.

7.1 Watertanks

As a first evaluation example, we use a well-known textbook example of a coupled water tank system [5], as illustrated in Fig. 9. 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 the form of a hybrid automaton. In this case study we focus on the extension or training part of our methodology and use a fixed BC.

Plant P. The plant consists of two tanks that are connected by a pipe that is located at a height of 0.2 me 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 Fig. 8. The automaton has two state variables x_1 and x_2, corresponding to 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 four locations reflect the two modes with different dynamics, the connected dynamic is applied to three locations, as otherwise the location would have a nonconvex invariant which is not supported by our tool. 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 dynamics.

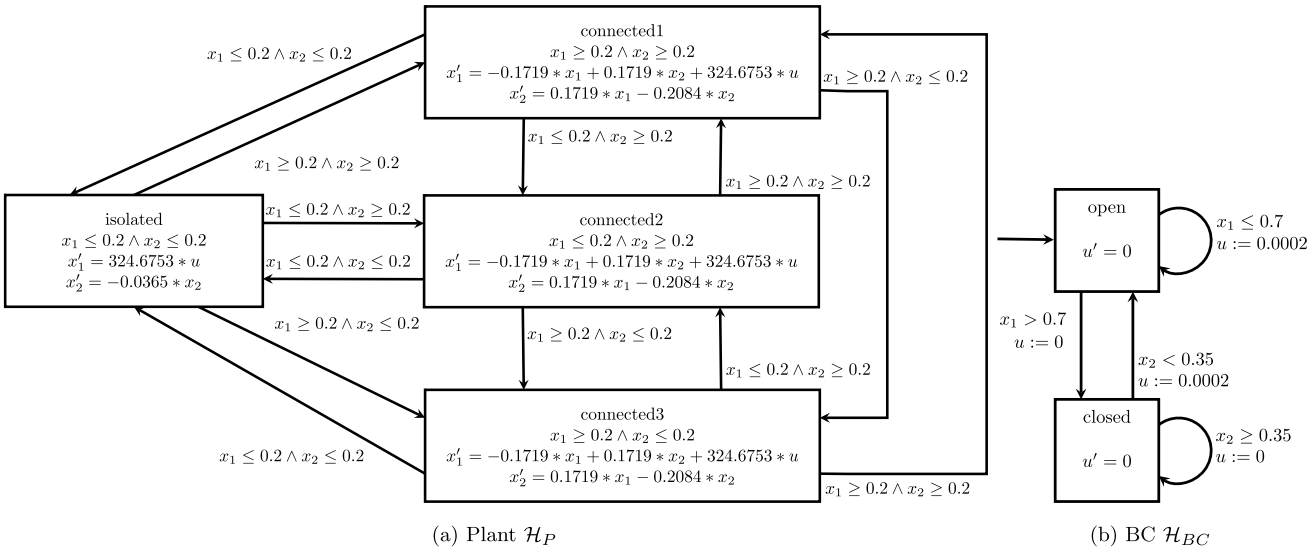
Safety specification φ. The safety specification $\varphi = \varphi_1 \wedge \varphi_2$ requires that the following two properties φ_1 and φ_2 are satisfied:

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 $\varphi_1 = \mathbf{G}\,(x_1 \leq 0.8 \wedge x_2 \leq 0.8)$.
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 $\varphi_2 = \mathbf{G}\left(x_2 \leq 0.12 \rightarrow \mathbf{F}_{[0,30]}\,(x_2 \geq 0.3)\right)$.

A safety specification φ in STL can be transformed into a hybrid automaton $\mathcal{H}_\varphi$ as described in Sect. 2.2.

Baseline controller. We use the baseline controller that is given by the hybrid automaton $\mathcal{H}_{BC}$ in Fig. 8. Here $\mathcal{H}_{BC}$ consists of two locations *open* and *closed*. At the end of a cycle ($t = 1$), the controller observes x_1 and x_2 to determine whether to stay in its current location or switch to the other. The baseline controller $\mathcal{H}_{BC}$ 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.

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 $\mathcal{H}_\varphi$ 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 $\mathcal{H}_\varphi$ is contained in the reachable states. We verified the baseline controller using the reachability analysis tool HyPro [25]. Figure 10(a) illustrates the initial recoverable region. The blue squares depict the initially known recoverable region. The orange trajectory



(a) Plant $\mathcal{H}_P$ (b) BC $\mathcal{H}_{BC}$

Fig. 8 Hybrid automaton model of the water tank plant $\mathcal{H}_P$ and the automaton $\mathcal{H}_{BC}$ implementing the baseline controller

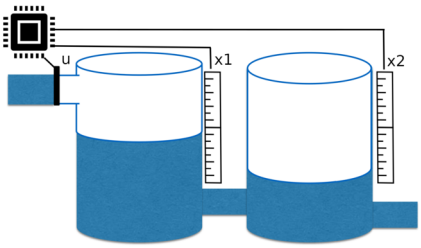


Fig. 9 Illustration of the water tank system

shows the behavior of the plant when controlled by the baseline controller, for a single point in the set of initial states. The red squares show the nodes of the hyperoctree data structure, used to allow efficient lookups in the recoverable region.

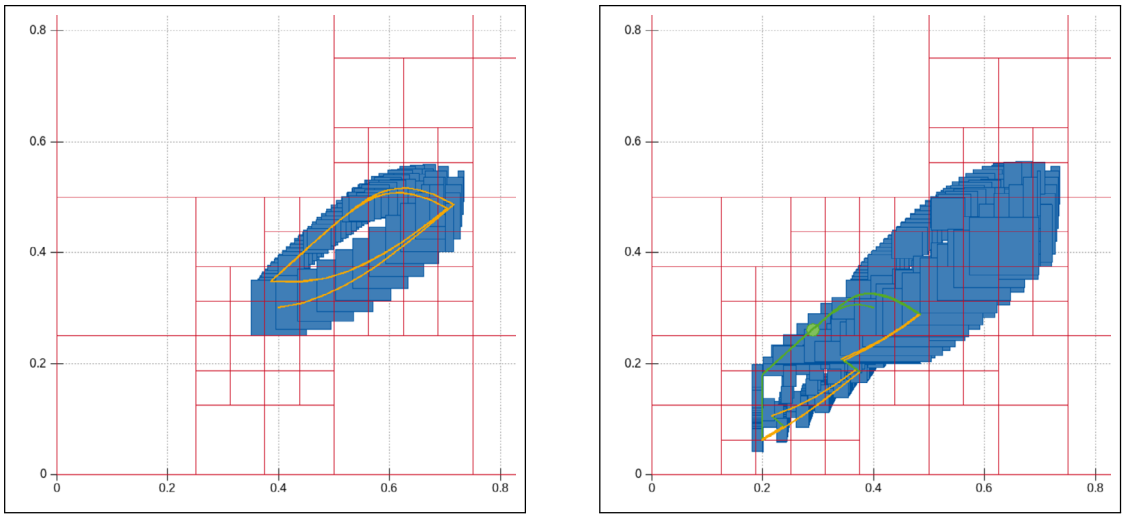
Adapting to the advanced controller via proof on demand. In the second step, we run the Simplex architecture for 500 control cycles and evolve the recoverable region during runtime. Figure 10(b)–(d) shows the growth of the recoverable region over time for different advanced controllers. In each figure, 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. Points where the trajectory exits the blue region 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 visits unsafe states that cannot be added to the recoverable region. The first advanced controller tries to minimize the amount of water in the tanks and sets u to 0 constantly. Figure 10(b) shows this result in an extension of S_r downwards, with the BC periodically intervening to ensure the bounded liveness specification. The second advanced controller (Fig. 10(c)) sets u to 0.0004 constantly to maximize the amount of water. This results in a few extensions for high values in the first tank and ultimately rapid switching between AC and BC with nearly full tanks. The third advanced controller (Fig. 10(d)) is used to show how our method works with chaotic AC actions. It picks $u \in [0.0001, 0.0005]$ equally distributed with a 10% probability, and sets $u = 0$ otherwise.

Results and observations. The results of our experiments allow for several interesting observations. Throughout the experiments, we recorded the set of safe regions along with trajectories (see Fig. 10), as well as data on the frequency of BC-invocations (in case the AC cannot guarantee safety) and the frequency of proofs on demand. Those data are summarized in Fig. 11.

After an initial analysis to determine a safe region based on the selected random initial state, the number of required proofs on demand over time generally decreases.

Our method shows a periodic behavior when using the constant zero controller. The water level x_2 drops below 0.12, but the AC does not attempt to raise it again. The BC intervenes at the last moment before the deadline can no longer be achieved since the switching logic (SL) prioritizes the AC as long as possible while maintaining satisfaction of the specification. This long period of the combined AC and BC results in slower convergence and an oscillation in the number of extensions depending on the state of the plant.

The constant controller which outputs 0.0004 quickly converges to zero proofs per 20 cycles. The number of BC invocations stabilizes at high values as the BC has to constantly



(a) initial region

(b) no inflow

(c) constant inflow

(d) random inflow

Fig. 10 Water tank case study: visualization of the initial recoverable region and adapted recoverable regions of the different advanced controllers for variables x_1, x_2 (Color figure online)

close the inflow valve to prevent an overflow. Even for the random controller, a saturation of the recoverable region could be observed; this can be attributed to the bounded randomness of the controller.

As expected, for both constant controllers, the BC is invoked periodically, since its output will only lead to unsafe behavior in a certain region of the state space, which is visited periodically due to the periodic nature of the system.

7.2 Autonomous racing car

We evaluate our approach using a controller for an autonomous racing car. This case study uses both verification extensions with a more complex BC and repair with a very simple initial BC.

The car is modeled as a point mass, observables are the position (x, y), the heading (θ), and its velocity (v). The car can be modeled by a hybrid automaton with a single location and nonlinear dynamics. For simplification, we allow instantaneous changes of the velocity and do not model acceleration. To obtain a *linear* hybrid system, we discretize θ and replace it with a representative such that the transcendental terms become constants (see Fig. 12a); the number of buckets for this discretization is parameterized and induces multiple locations (one for each bucket for each discretized variable, see Fig. 12b). The experiments are done with a discretization of 36 θ-buckets.

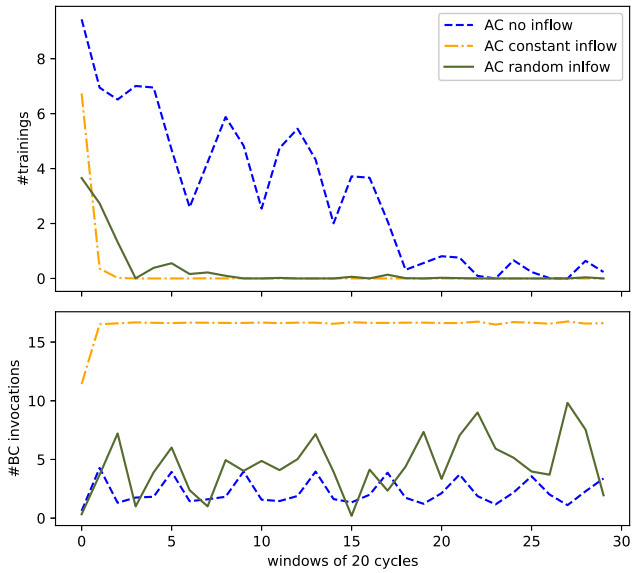
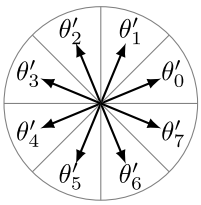
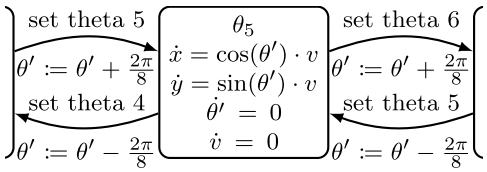


Fig. 11 Successful extensions of S_r per 20 cycles for the water tanks system. The plot shows the average from 200 random initial states



(a) The center of a bucket gives θ' as a representative for that bucket.



(b) Locations for different buckets are only connected by controller-actions (label synchronization), which are periodically enabled with cycle time δ.

Fig. 12 Modeling approach for the discretized car for 8 buckets

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 Fig. 13a), 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. The curbs $P_{curbs} \subset P_{in}$ are defined as the left and right 10% of the road. Formally, the specification is $\varphi = \mathbf{G}\left((x,y) \in P_{in}\right) \wedge \mathbf{G}\left((x,y) \notin P_{out}\right) \wedge \mathbf{G}\left(\left((x,y) \in P_{curbs}\right) \rightarrow \mathbf{F}_{[0,2]}\left((x,y) \notin P_{curbs}\right)\right)$.

Baseline controller. To model the baseline controller, each track is subdivided into an ordered sequence of straight segments. The actions of the baseline controller drive the car to the center line of the current segment, turn it parallel to the road, and stop. 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 [3] that is equipped with a set of waypoints along the track. Waypoints are either given by points in the middle of the track between two segments (AC 1) or obtained by a race line optimizer tool [9] (AC 2).

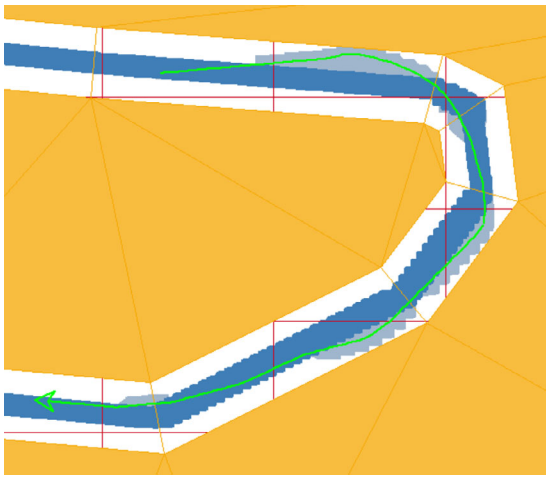
Results and observations. For evaluation, we consider linearizations of two F1 racetracks: Spielberg in Austria and Silverstone in the UK.

The 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_r during a single training run. To show the influence of proofs on demand, we synthesized S_r for the center area of a whole track a priori.

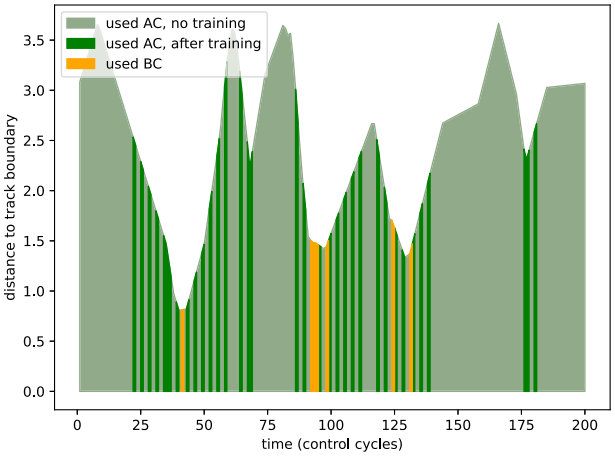
Depending on the selection of waypoints for the advanced controller, naturally, the usefulness of the initial S_r varies. AC 1 with waypoints in the middle of the road tends to induce fewer extensions of S_r than the set of waypoints generated by the raceline optimizer (AC 2), which selects points allowing a path with less curvature that can get very close to the edge of the road.

Figure 13 shows how online verification is performed for a small segment of the Spielberg race track. Figure 13a visualizes the optimized trajectory, S_r, as well as its extensions. We recorded the invocations of adaptive proofs over time and the distance to the boundary in Fig. 13b. The car starts at the center of the road and thus far away from the boundary. During the first few control cycles the car remains in the verified region (the dark blue area in Fig. 13a), but reduces its distance to the boundary. As can be seen in Fig. 13b, the early cycles use the advanced controller without additional verification. Once the car exits the preverified region training is performed to keep using the advanced controller. The extensions of the verified region are shown in light blue in Fig. 13a and the cycles where an extension is performed are marked as dark green in Fig. 13b. When the car gets to close to the boundary, the dynamic verification fails and the baseline controller has to be used to steer the car away from the boundary (orange in Fig. 13b). The remainder of the turn is also shown in Fig. 13.

Figure 14 shows the results for ten rounds on the Spielberg and Silverstone race tracks. Controllers based on two sets of waypoints are used, where AC 1 corresponds to the



(a) The car's trajectory is shown in green. Leaving S_r (dark blue) triggers extension thereof (light blue).



(b) Distance to the closest track boundary over time. Controller usages are color-coded for each control cycle.

Fig. 13 Execution of the race car with online verification (Color figure online)

Table 1 Bloatings used for proofs on demand affect the success of this approach (15000 cycles/ 7.5 laps, no initial recoverable region, only verification extension)

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

standard waypoints and AC 2 corresponds to the optimized waypoints. The results for the repair method (AC 1 repair) are discussed in the next section. We can observe, that initially many proofs are required to saturate S_r in the first lap. Later laps require fewer adaptations, and 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 so that evolving S_r is necessary to use the advanced controller as often as possible. In the long run, the number of proofs per lap is expected to reach zero. The initial position used to generate proofs on demand is bloated to allow larger extensions of S_r at a time. Our experiments with different bloating parameters (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. This observation is in line with our expectations, as attempts for more aggressive extension of S_r due to larger bloating may lead, if successful, to fewer requests for further extensions. Otherwise, if those attempts are not successful, the consequence is more invocations of the baseline controller. Selecting a very large bloating parameter can result in a situation where the stop action is actually executed because the suggested action can not be proven safe. A user can choose an appropriate trade-off between verification time and how conservative the safety check is.

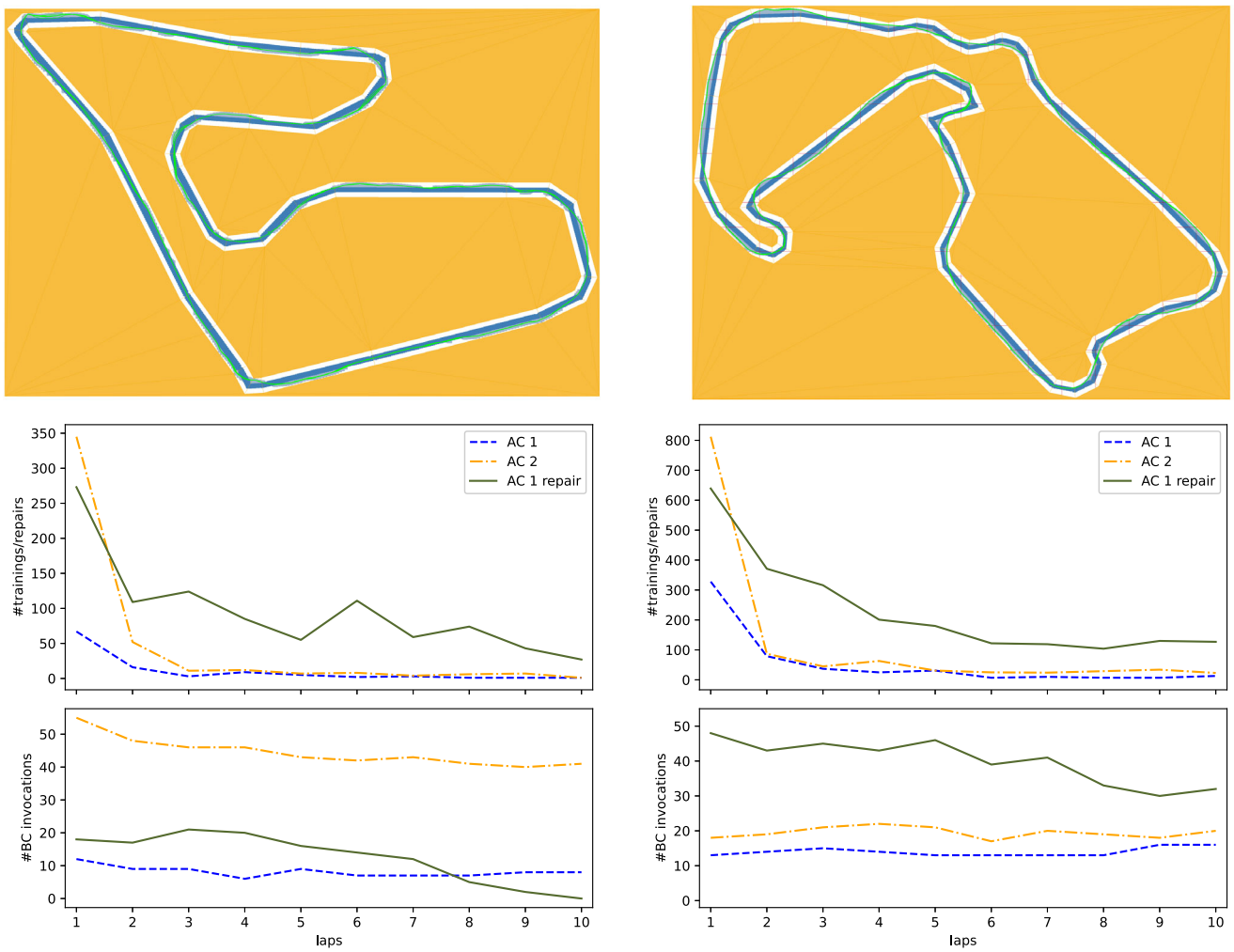
On a standard desktop computer, the computation of the initial recoverable region takes about 20 minutes, performing a proof on demand to extend the recoverable region takes less than a second, and testing whether to switch without extending takes about 10 milliseconds.

7.2.1 Repair

To evaluate our BC repair method we adapted the racetrack examples. Instead of using a complete BC, only some clearly safe situations are defined, and the remaining BC and recoverable region are synthesized. The initial BC consists of the states within 1.0 units of the center line with a heading that matches the orientation of the road. All of these states are associated with the stop action. This controller trivially satisfies the requirement that the car remains inside the track for infinite time, for all states where it is defined. Starting with the simplex architecture obtained from this BC we use adaptive repair to improve the BC and extend the recoverable region to allow for better performance of the AC.

Patch templates. We use patch templates and enumeration to generate patch candidates for states that are not handled correctly by the BC. The template is general enough to apply to most driving scenarios while eliminating nonsensical driving patterns such as rapidly switching between two headings.

Each patch candidate consists of the following sequence: turn to heading h_1 and continue for t_1 cycles then turn to



(a) Spielberg (b) Silverstone

Fig. 14 Successful extensions of S_r (top: blue) per lap for the Spielberg and Silverstone race tracks. Starting from a moderately large known recoverable region for two sets of waypoints (standard = AC 1, optimized = AC 2) using verification training and using repair for the standard waypoints (Color figure online)

heading h_2 and continue for t_2 cycles. The car can only turn one theta bucket in each control cycle. When changing the heading, the car turns one step towards the target heading each cycle and if there is time remaining it continues to follow the target heading. The possible parameters for the template parameters are every third heading for h_1 and h_2, the values 1, 6, 11, 16 for t_1, and $t_2 = 5$. This template can generate most U- and S-shaped turns.

Results and observations. The results of applying our repair methodology to the two race tracks can be seen in Fig. 14. We can observe similar results as with verification extensions. The first few laps need a large number of repair extensions. However, once the recoverable region has adapted to the AC, the number of repairs is greatly reduced. Compared to a given BC, this decreases slower and requires more adaptations. The number of BC invocations decreases slightly in the later laps as fewer interventions happen due to an insufficiently large recoverable region. As can be expected, repairing a BC in this way takes significantly more time than verifying an existing BC. A simulation with online repair of 10 laps took 4 h for the Spielberg track and 18 h for the Silverstone track. The repair methodology should thus be used in combination with a simulator which is used to compute an adapted BC and recoverable region for a given AC, before deploying the resulting simplex architecture on the real system.

Using automatic repair of the BC allows expending more computation time to obtain comparable AC performance while spending less time designing a baseline controller.

8 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 to the behavior of the running advanced controller, by using verification extensions and automatic repair. 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 program under grant agreement № 956123 – FOCETA, the Austrian research promotion agency FFG projects ADVANCED (№ 874044) and FATE (№ 894789), the TAIGER WWTF project ICT22-023, the Graz University of Technology LEAD Project *Dependable Internet of Things in Adverse Environments*, and the State Government of Styria, Austria – Department Zukunftsfonds Steiermark. This research was funded in whole, or in part, by the Austrian Science Fund (FWF) 10.55776/ZK-35. For the purpose of open access, the author has applied a CC BY public copyright licence to any Author Accepted Manuscript version arising from this submission.

Funding Open access funding provided by Graz University of Technology.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Alshiekh, M., Bloem, R., Ehlers, R., Könighofer, B., Niekum, S., Topcu, U.: Safe reinforcement learning via shielding. In: AAAI, pp. 2669–2678. AAAI Press, Menlo Park (2018)
2. Alur, R., Courcoubetis, C., Henzinger, T.A., Ho, P.H.: Hybrid automata: an algorithmic approach to the specification and verification of hybrid systems. In: Hybrid Systems, pp. 209–229. Springer, Berlin (1992). https://doi.org/10.1007/3-540-57318-6_30
3. Amidi, O., Thorpe, C.E.: Integrated mobile robot control. In: Mobile Robots V, vol. 1388, pp. 504–523. SPIE, Bellingham (1991). https://doi.org/10.1117/12.25494
4. Bak, S., Manamcheri, K., Mitra, S., Caccamo, M.: Sandboxing controllers for cyber-physical systems. In: ICCPS, pp. 3–12. IEEE Comput. Soc., Los Alamitos (2011). https://doi.org/10.1109/ICCPS.2011.25
5. Belta, C., Yordanov, B., Aydin Gol, E.: Formal Methods for Discrete-Time Dynamical Systems. Springer, Berlin (2017). https://doi.org/10.1007/978-3-319-50763-7
6. Chutinan, A., Krogh, B.H.: Computational techniques for hybrid system verification. IEEE Trans. Autom. Control **48**(1), 64–75 (2003). https://doi.org/10.1109/TAC.2002.806655
7. Crenshaw, T.L., Gunter, E.L., Robinson, C.L., Sha, L., Kumar, P.R.: The simplex reference model: limiting fault-propagation due to unreliable components in cyber-physical system architectures. In: RTSS, pp. 400–412. IEEE Comput. Soc., Los Alamitos (2007). https://doi.org/10.1109/RTSS.2007.34
8. Frehse, G., Kekatos, N., Nickovic, D., Oehlerking, J., Schuler, S., Walsch, A., Woehrle, M.: A toolchain for verifying safety properties of hybrid automata via pattern templates. In: ACC, pp. 2384–2391. IEEE (2018). https://doi.org/10.23919/ACC.2018.8431324
9. Heilmeier, A., Wischnewski, A., Hermansdorfer, L., Betz, J., Lienkamp, M., Lohmann, B.: Minimum curvature trajectory planning and control for an autonomous race car. Veh. Syst. Dyn. **58**(10), 1497–1527 (2020). https://doi.org/10.1080/00423114.2019.1631455
10. Henzinger, T.A.: The theory of hybrid automata. In: Verification of Digital and Hybrid Systems, pp. 265–292. Springer, Berlin (2000). https://doi.org/10.1007/978-3-642-59615-5_13
11. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? J. Comput. Syst. Sci. **57**(1), 94–124 (1998). https://doi.org/10.1006/jcss.1998.1581
12. Ionescu, T.B.: Adaptive simplex architecture for safe, real-time robot path planning. Sensors **21**(8), 2589 (2021). https://doi.org/10.3390/s21082589
13. Johnson, T.T., Bak, S., Caccamo, M., Sha, L.: Real-time reachability for verified simplex design. ACM Trans. Embed. Comput. Syst. **15**(2), 26:1–26:27 (2016). https://doi.org/10.1145/2723871
14. Maderbacher, B., Schupp, S., Bartocci, E., Bloem, R., Nickovic, D., Könighofer, B.: Provable correct and adaptive simplex architecture for bounded-liveness properties. In: Model Checking Software – 29th International Symposium, SPIN 2023, Proceedings, Paris, France, April 26-27, 2023, pp. 141–160 (2023). https://doi.org/10.1007/978-3-031-32157-3_8
15. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: FORMATS/FTRTFT. Lecture Notes in Computer Science, vol. 3253, pp. 152–166. Springer, Berlin (2004). https://doi.org/10.1007/978-3-540-30206-3_12
16. Marta, D., Pek, C., Melsión, G.I., Tumova, J., Leite, I.: Human-feedback shield synthesis for perceived safety in deep reinforcement learning. IEEE Robot. Autom. Lett. **7**(1), 406–413 (2022). https://doi.org/10.1109/LRA.2021.3128237
17. Meagher, D.: Geometric modeling using octree encoding. Comput. Graph. Image Process. **19**(2), 129–147 (1982). https://doi.org/10.1016/0146-664X(82)90104-6
18. Mehmood, U., Stoller, S.D., Grosu, R., Roy, S., Damare, A., Smolka, S.A.: A distributed simplex architecture for multi-agent systems. In: SETTA. Lecture Notes in Computer Science, vol. 13071, pp. 239–257. Springer, Berlin (2021). https://doi.org/10.1007/978-3-030-91265-9_13
19. Mehmood, U., Stoller, S.D., Grosu, R., Smolka, S.A.: Collision-free 3D flocking using the distributed simplex architecture. In: Formal Methods in Outer Space. Lecture Notes in Computer Science, vol. 13065, pp. 147–156. Springer, Berlin (2021). https://doi.org/10.1007/978-3-030-87348-6_9

20. Phan, D.T., Grosu, R., Jansen, N., Paoletti, N., Smolka, S.A., Stoller, S.D.: Neural simplex architecture. In: NFM. Lecture Notes in Computer Science, vol. 12229, pp. 97–114. Springer, Berlin (2020). https://doi.org/10.1007/978-3-030-55754-6_6
21. Prajna, S., Jadbabaie, A.: Safety verification of hybrid systems using barrier certificates. In: HSCC. Lecture Notes in Computer Science, vol. 2993, pp. 477–492. Springer, Berlin (2004). https://doi.org/10.1007/978-3-540-24743-2_32
22. Romdlony, M.Z., Jayawardhana, B.: Stabilization with guaranteed safety using control Lyapunov-barrier function. Automatica **66**, 39–47 (2016). https://doi.org/10.1016/j.automatica.2015.12.011
23. Schupp, S.: State Set Representations and Their Usage in the Reachability Analysis of Hybrid Systems. PhD thesis, RWTH Aachen University, Aachen (2019). https://doi.org/10.18154/RWTH-2019-08875
24. Schupp, S., Ábrahám, E., Chen, X., Makhlouf, I.B., Frehse, G., Sankaranarayanan, S., Kowalewski, S.: Current challenges in the verification of hybrid systems. In: International Workshop on Design, Modeling, and Evaluation of Cyber Physical Systems, pp. 8–24. Springer, Berlin (2015). https://doi.org/10.1007/978-3-319-25141-7_2
25. Schupp, S., Ábrahám, E., Makhlouf, I.B., Kowalewski, S.: HyPro: a C++ library of state set representations for hybrid systems reachability analysis. In: NFM. Lecture Notes in Computer Science, vol. 10227, pp. 288–294 (2017). https://doi.org/10.1007/978-3-319-57288-8_20
26. Seto, D., Krogh, B., Sha, L., Chutinan, A.: The simplex architecture for safe online control system upgrades. In: ACC, pp. 3504–3508. IEEE (1998). https://doi.org/10.1109/ACC.1998.703255
27. Sha, L.: Using simplicity to control complexity. IEEE Softw. **4**, 20–28 (2001). https://doi.org/10.1109/MS.2001.936213
28. Shivakumar, S., Torfah, H., Desai, A., Seshia, S.A.: SOTER on ROS: a run-time assurance framework on the robot operating system. In: RV. Lecture Notes in Computer Science, vol. 12399, pp. 184–194. Springer, Berlin (2020). https://doi.org/10.1007/978-3-030-60508-7_10
29. Simão, T.D., Jansen, N., Spaan, M.T.J.: Alwayssafe: reinforcement learning without safety constraint violations during training. In: Dignum, F., Lomuscio, A., Endriss, U., Nowé, A. (eds.) AAMAS'21: 20th International Conference on Autonomous Agents and Multiagent Systems, Virtual Event, United Kingdom, May 3–7, 2021, pp. 1226–1235. ACM, New York (2021). https://doi.org/10.5555/3463952.3464094
30. Yang, J., Islam, M.A., Murthy, A., Smolka, S.A., Stoller, S.D.: A simplex architecture for hybrid systems using barrier certificates. In: SAFECOMP. Lecture Notes in Computer Science, vol. 10488, pp. 117–131. Springer, Berlin (2017). https://doi.org/10.1007/978-3-319-66266-4_8

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.